Einführung in die Programmierung

Übersicht zur Lehrveranstaltung

Benotung

Angekreuzte Aufgaben = 40%

Mindestens **75% muss angekreuzt** sein!

Wöchentliche Prüfungen = 30%

Mindestens 50% müssen erreicht werden.

Midterm Test = 30%

Eine zufriedenstellende Präsentation erforderlich.

Midterm-Test

Ähnlicher Modus wie VO-Klausur. Findet am PC statt und besteht zu einem Teil aus theoretischem Wissen und zum anderen aus praktischen Wissen.

Termin: **13. Dezember, 17:45 bis 19:45**

Inhaltsverzeichnis

Einleitung Allgemeine Programmierung

Grundsätzlich löst ein Computer Probleme, indem er Instructions (Befehle) der Reihenfolge abarbeitet. Ein Programm stellt daher einen bestimmten Satz von Befehlen dar.

Da ein Computer durch elektronische Schaltungen realisiert wird, kann er nur eine endliche Menge an einfachen Befehlen ausführen (welche selbst zwar Binär kodiert sind (z.B. MIPS 32-Bit Architektur), aber direkt in Assembly umgewandelt werden kann).

Da selbst Assembly noch scheiße schwer zu lesen ist, wurden höhere Programmiersprachen entworfen. Wesentlich wichtiger für diesen Schritt war jedoch die Tatsache, dass verschiedene Maschinen verschiedene Maschinensprachen hatten. D.h. man musste eine Sprache erzeugen, die beim Kompilieren auf die jeweilige Zielplattform übersetzt.

IDEs

Sind Entwicklungsumgebungen, die das Programmieren vereinfachen, wie z.B. **Eclipse CDT** oder JetBrains CLion.

The C-Programming Language

Entworfen von Dennis Ritchie und Ken Thompson 1969, gab es 1978 das erste Buch von Brian Kernighan und Dennis Ritchie, welche die C-Sprache erläuterte und definierte.

TIPP: Ich empfehle wirklich „The C-Programming Language“ von Kernighan and Ritchie!
Ich selbst lese kaum Fachliteratur, aber dieses Buch ist klein und beinhaltet das umfassende Wissen über C!

1999 wurde dann der C99 Standard entworfen, der in der Vorlesung verwendet wird.

Eigenschaften von C

C ist eine imperative, also welche Anweisungen werden in welcher Reihenfolge vom Computer ausgeführt, und strukturierte Programmiersprache. Sie ermöglicht direkte Speicherzugriffe und damit eine sehr hardwarenahe Programmierung.

Wenige Schlüsselwörter, die dennoch äußert mächtig verwendet werden können.

C ist die Grundlage vieler höhere Programmiersprachen (Java, C++, C#, Python, ...) und C-Compiler gibt’s auf nahezu allen Plattformen.

Kompilierung

Grundsätzlich gibt es Source-Code-Dateien und Header-Dateien. Letztere beinhalten Informationen, die man für das Kompilieren braucht und geben dem Developer zudem einen Überblick über definierte Datentypen und Funktionen im Programm.

GCC-Compiler

Der GCC-Compiler ist grundsätzlich auf jedem Linux-System verfügbar. Auf Windows bietet sich der MinGW an.

In der Vorlesung verwenden wir die Konvention

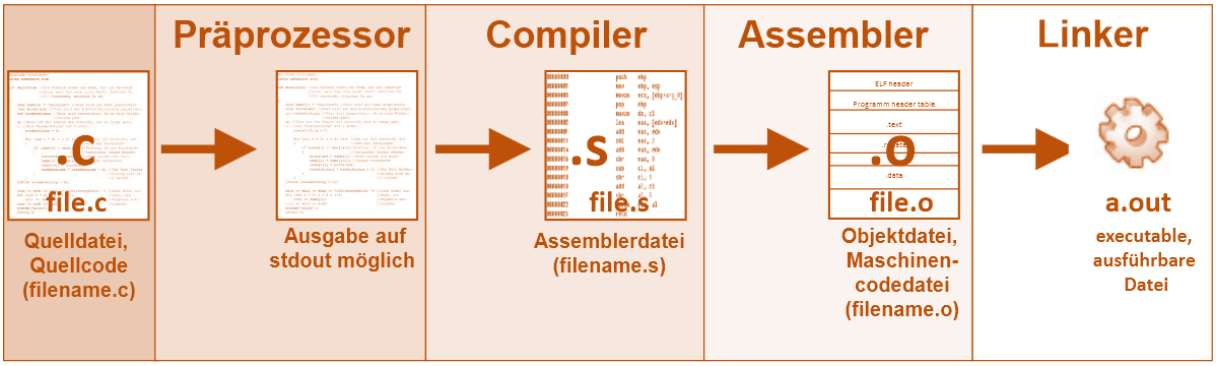
```
gcc -Wall -Werror -std=c99 program.c -o program
```

Damit lassen sich unsere einfachen Programme kompilieren.

Mittels

```
gcc program.c -c program.o
```

Lassen sich die Objekt-Dateien erzeugen, welche dann vom Linker zum Programm übersetzt werden würden.



Aufrufmöglichkeiten

Wollen wir mehrere Dateien kompilieren und zu einem Programm linken, können wir dies relativ simpel mittels

```
gcc -o program file1.c file2.c
```

Wollen wir das ganze Schritt für Schritt machen, können wir den -c Parameter nutzen:

```
gcc -c file1.c //creates file1.o
gcc -c file2.c //creates file2.o
gcc -o program file1.o file2.o
```

Wir kompilieren die Dateien und linken sie mit dem letzten Aufruf.

Suffixe

file.c	C-Quelldatei, sie muss Präprozessor durchlaufen
file.i	C-Quelldatei, darf die Präprozessorphase nicht durchlaufen
file.s	Assemblerdatei
file.o	ObjectDatei
a.out	Ausführbare Datei.

Allgemeine Flags

-E Stoppt nach dem Aufruf des Präprozessors, der Compiler wird nicht aufgerufen. Die Ausgabe davon wird an stdout (Konsole) geschickt.

-S Stoppt nach dem Übersetzer. Der Assembler wird nicht aufgerufen und eine Datei mit dem Suffix .s wird erzeugt.

-c Quellprogramm durchläuft Präprozessor, Compiler und Assembler. Der Linker aber wird nicht ausgeführt und die erzeugte Datei hat den Suffix .o Das ELF (Executable und Linkable Format) beschreibt das Stadard-Binärformat ausführbarer Programme unter UNIX.

-o filename Output (z.B. Executable) wird in die Datei filename transferiert. Standardmäßig wird nach a.out geschrieben.

-x language Spezifiziert die Programmiersprache, welche normalerweiße basierend auf der Dateiendung gewählt wird. Diese Option ist bis zum nächsten Aufruf von -x dauerhaft gültig!! -x none setzt es zurück.

-v Verbose-Mode, Zeigt am stderr detailierte Infos mit welchen Optionen GCC arbeitet.

--version Zeigt die Version der GCC am stdout an

--help Joa, was wohl.

C-Language specific options

Für C gibt es eigene Flags, die GCC annimmt:

-ansi spezifiziert den zugrundeliegenden ISO C Standard, der 1990 eingereicht wurde. Äquivalent zu -std=c90

-std=c99 spezifiziert den ISO C Standard vom Dezember 1999.

Warning Options

-Wall Diese Option schaltet einen Großteil sehr nützlicher Warnungen für den täglichen Gebrauch ein.

-Wextra Schaltet zusätzlich noch Warnungen ein, die von -Wall nicht berücksichtigt wurden.

-Wpedantic Woaaah, das zeigt alles an. Siehe manpage

-Werror Wandelt Warnungen zu Fehler um.

Debugging Options

-g Erzeugt Debugger-Informationen für das jeweilige Betriebssystemabhängige Fileformat. GDB (GNU Debugger) kann mit diesen zusätzlichen Infos umgehen, andere könnten Probleme haben.

-g*Level* Ermöglicht die Einstellungen des Debugging Levels, Std=2
Level 0, keine Infos (-g0 negiert -g)
Level 1, minimale Infos
Level 2, zusätzliche Infos, standard.

-ggdb Debugger-Infos speziell für den GDB mit passenden Format.

Optimization Options

-O0 Reduziert Übersetzungszeit und stellt Debugging-Infos ohne Veränderung des Programmcodes durch Optimierungen zur Vefügung, ist der Standard.

-O1 / -O Optimierungen werden vorgenommen, welche die Programmgröße und Ausführungszeit optimieren. Übersetzung braucht mehr Zeit und deutlich mehr Hauptspeicher!

-O2 Verwendet alle -O1 Optimierungen, inklusive weiteren, wie bessere Performance, längere Übersetzungszeit. Achtung, hier wird manchmal der Effekt von schlechten Code verändert!

-O3 Alles von O2 + aggressive Optimierungsstrategien. Bringt selten was.

-Os Optimiert die Größe des Programmcodes. Verwendet alles von O2, was nicht die Größe verändert und Optimierungen, welche den Code verkleinern.

GNU Binutils

Ist eine Sammlung von Werkzeugen, um Binär- und Objektdateien zu linken, manipulieren und in Maschinencode umzuwandeln.

Enthalten sind:

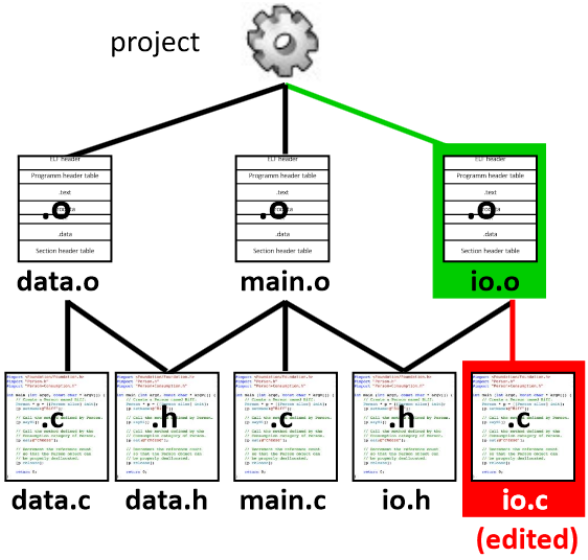
- as Assembler
- gprof Profiler
- ld Linker
- nm Zeigt Symbole einer Objektdatei
- objdump Zeigt Infos über Objektdateien
- readelf Zeigt Infos über ELF-Dateien
- size Zeigt Größe einzelner Abschnitte
- strings Zeigt druckbare Strings
- strip Entfernt Symbole aus Objektdatei
- addr2line, ar, c++filt, gold, objcopy (und mehr)

GNU MAKE

Ermöglicht das Management von Übersetzungsvorgängen und von Abhängigkeiten in großen Programmen. Kann zum Übersetzen, (De-)Installieren, Unit-Tests, ... verwendet werden.

make wird über einfache Textdatei, sprachenunabhängig gesteuert. MAKEFILEs sind portable, leicht wiederverwendbar und können den Build-Prozess sehr einfach machen.

Der große Vorteil von make ist unter anderem der Abhängigkeitsgraph, der nur jeden Dateien übersetzt, die auch wirklich geändert wurden:



Das Makefile

Kann aus **Variablendefinitionen, Anweisungen, Explizite Regeln, Implizite Regeln** und **Kommentaren** bestehen.

Der Aufruf erfolgt über die Konsole mittels make, wobei das aktuelle Verzeichnis nach „Makefile“ oder „makefile“ (manchmal auch „GNUMakefile“) durchsucht wird. Ist es vorhanden, erzeugt make auch gleich den Abhängigkeitsgraphen mithilfe von Regeln, die Abhängigkeiten beschreiben:

```
Ziel(e): Vorbedingung(en)
        Kommando(s) #Tabs verwenden!!
```

Dabei werden die Kommandos nur durchgeführt, wenn die Vorbedingungen neuer als die Ziele sind oder diese nicht vorhanden sind.

Vorbedingungen

Können auch andere Ziele anderer Regeln sein. Make prüft immer, ob die Vorbedingung vorhanden sind und aktualisiert diese gegebenenfalls. Erst wenn alle Vorbedingungen aktuell sind, werden die Kommandos abgearbeitet.

Regeln/Ziele

Die Reihenfolge der Regeln ist wurscht, nur das erste ist das Standardziel, wenn beim Aufruf von make nichts anderes angegeben wird.

Unechte Ziele

Sogenannte **unechte Ziele** sind jene Regeln, bei denen das Ziel kein Dateiname ist. Typisch dafür ist z.b. „clean:“, welches für Aufräumarbeiten genutzt wird. Für solche unechten Ziele sollte .**PHONY** verwendet werden, um Überschneidungen mit einer möglichen Datei clean und bessere Performance zu erzielen.

```
.PHONY: clean
clean:
	rm *.o *.h~ *.c~ lib*.*
```

Musterregeln

Musterregeln hingegen sind Muster, in dem das Zeichen **%** als Platzhalter für beliebig viele, nicht leere Zeichen stetht.

```
%.o: %.c %.h
	Kommandos
```

Damit lassen sich defacto alle *.o bauen, wenn *.c und *.h gegeben sind.

Impliziten Regeln

Zuletzt die **Impliziten Regeln**, sind von make vordefinierte Suffix- und Musterregeln. **make -p** zeigt diese Regeln.

Variablen

Machen Makefiles übersichtlich und portable, leicht wartbar. Erlaubte Werte sind z.B.: Dateinamen, Liste von Dateinamen, Compiler-Schalter, Kommandos, etc.

Variablen von Makefiles sind rekursiv expandierend, das heißt, wenn der Wert eine andere Variable enthält, wird diese zuerst ausgeführt/ersetzt. Mit
Bezeichner := Wert wird der Wert sofort expandiert.

Der Zugriff erfolgt mittels $(Bezeichner), wobei dies nur eine textuelle Ersetzung ist.

Make ist auch in der Lage, Variablen in der Kommandozeile zu übergeben

```
make CC:=gcc
```

Automatische Variablen

Werden von make automatisch beim Abarbeiten gesetzt und können von Kommandos verwendet werden:

$@ : Enthält den Namen des Ziels
$< : Name der ersten Vorbedingung
$? : Namen aller Vorbedingungen, die neuer als das Ziel sind
$^ : Namen aller Vorbedingungen, ohne Duplikate
$+ : Namen aller Vorbedingungen, mit Duplikate
$* : Übereinstimmender Namensstamm einer Regel
$(@D) liefert directory
$(@F) liefert file.c

Vordefinierte Variablen

CC	cc
CFLAGS	-
LD	ld
LDFLAGS	-
RM	rm -f

Definitionen und Erklärungen

Deklaration:
umfasst Namen und Typ einer Variable und macht dem Compiler bekannt, mit welchem Typ eine Variable verbunden wird. Also Datenobjekte bekannt machen

Definition:
Ist die Deklaration und die Reservierung des Speicherplatzes

Initialisierung:
Ist das erstmalige „befüllen“ der Speicheradresse. Davor steht noch der alte Wert drin.

Ausdruck: (expression)
Kombination von Operatoren und Operanden, damit kann ein Ausdruck ein Teil eines größeren Ausdrucks sein.

Anweisung: (statement)
Wenn ein Semikolon am Ende steht, ist es eine Anweisung, diese haben keinen Wert, sind kein Teil eines größeren Ausdrucks: i++;

Prozedurale Programmierung

Ein Algorithmus wird in überschaubare Teile zerlegt, die über definierte Schnittstellen aufrufbar sind. Sie erweitert das imperative Paradigma als Folge von definierten Zustandsübergängen, bei denen festgelegt ist, wie Zustände verändert werden sollen.

Es besteht kein Zusammenhalt zwischen Daten und Funktionen!

ANSI

Der ANSI-Standard war sozusagen der erste richtige Standard, nach ein paar Monaten (oder Jahren, ka) Programmierung und Nutzung der C-Sprache. Man hat

wichtige Erneuerungen und Erweiterungen eingeführt, um C wirklich in der Praxis verwenden zu können. So ist zum Beispiel auch die Standardbibliothek mit dem ANSI-Standard eingeführt worden. Diese Standardbibliothek war (und ist) **kein** Teil der C-Sprache.

Standard-Bibliothek

Ich hab dieses Kapitel von unten vorgeschoben, weil es wichtig ist, dies als Grundlage kennen zu lernen, also vorerst einfach mal getrost so hinnehmen – wird dann später klar was damit gemeint ist.

Eine Auswahl wichtiger Header-Dateien:

- stdio.h Ein- und Ausgabe
- ctype.h Tests für Zeichenklassen
- string.h Funktionen für Zeichenketten
- math.h Mathematische Funktionen
- stdlib.h Hilfsfunktionen: Umwandlung von Zahlen, Speicherverwaltung
- time.h Funktionen für Datum und Uhrzeit
- limits.h Definiert Konstanten für den Wertumfang der ganzzahligen.
- float.h Definiert Konstanten die sich auf Gleitpunktarithmetik beziehn

Die C-Programmierung

Kommentare

Dienen zur zusätzlichen Erklärung. Bitte niemals erklären, was dein Code tut, weil jeder Programmierer lesen kann, was dein Code tut. Beschreibe stattdessen, warum dein Code es tut, warum genauso und füge hilfreiche Informationen dazu.

/* ... */ Ist ein Ein-/ & Mehrzeiliger Kommentar

// ... dagegen ist nur ein Einzeiliger Kommentar, der defacto nur in c99 definiert ist, aber mittlerweile eh von jedem Compiler verwendet wird. Aber er könnte Probleme beim Portieren bereiten.

Präprozessor-Direktiven

Der Präprozessor arbeitet vorm kompilieren seine Aufgaben ab (deswegen Prä...).

Jede Präprozessordirektive beginnt mit einer Raute # und dem Befehl. Das Resultat daraus wird dann dem Compiler übergeben.

#Include

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

Include bindet eine Datei ein. Grundsätzlich wird damit der komplette Inhalt der Datei einfach an diese Stelle eingefügt.

#Define

```
#define NAME VALUE
```

Damit kann man Werte übersichtlich definieren. Der Präprozessor fügt dann an jeder Stelle wo NAME verwendet wird den VALUE ein. Damit sind diese Werte dann

auch Immediate-Werte, keine tatsächlichen „C-Konstanten“ bzw. auch keine Variablen. Sie sind unheimlich praktisch, weil sie zentralisiert und erklärend sind.

Variablen

Variablen werden an eine bestimmte Speicheradresse gespeichert, abhängig vom Typ der Variable mit einer bestimmten Länge. Sie ist also lediglich eine benannte Speicheradresse. Der Typ legt auch fest, welche Operationen darauf angewandt werden dürfen.

Deklaration und Definition solltest alleine schaffen, sonst kannst die gleich aufhängen gehen. Ist eine Variable deklariert, aber nicht initialisiert, beinhaltet sie einen zufälligen Wert, eben jenen, der davor in der Speicherstelle stand.

Variablen müssen mit einem Buchstaben oder Unterstrich beginnen, dürfen Zahlen, Buchstaben und Unterstriche beinhalten und sind Case-Sensitive.

Elementare Datentypen

Char
Speichert ein Zeichen oder eine Ganzzahl, wobei dieser immer als Zeichen interpretiert wird.

Int
Für ganzzahlige Zahlen mit Vorzeichen

Float
Für Gleitkommazahlen mit einfacher Genauigkeit

Double
Für Gleitkommazahlen mit doppelter Genauigkeit

Typmodifikatoren

Die obigen Datentypen können durch Typmodifikatoren angepasst werden:

Short int
Ganzzahliger Datentyp, der maximal den Wertebereich von int besitzt (also kleiner ist), wobei das int weggelassen werden kann.

Long int
Ganzzahliger datentyp, der mindestens den Wertebereich von int besitzt (also größer ist), wobei int weggelassen werden kann.

Long double
Gleitkommazahl mit erweiterter Genauigkeit, je nach System identisch zu double

Long long int
Erweiterter ganzzahliger Typ, kann identisch zu long sein; int kann weggelassen werden.

Signed / Unsigned
Char und alle ganzzahligen Datentypen können bezgl. dem Vorzeichen modifiziert werden. Dazu wird einfach signed oder unsigned vor den Datentyp vorangestellt.

Unsigned: Das Vorzeichenbit wird für den Zahlenwert freigegeben
Signed: Ein Bit wird für die Darstellung des Vorzeichens reserviert. (Standard)

Größe der Datentypen

Diese sind immer abhängig vom System, wobei der C-Standard die minimale Größe angibt.

Literale

Eine Zahl, Zeichenkette oder Wahrheitswert direkt im Quellcode, nennt man Literal. Dabei kann man Dezimal, Oktal oder Hexadezimal laden, mit **0**, **0.** oder **0x**.

Mit einem Suffix kann der Datentyp des Literals bestimmt werden: **u** (unsigned), **l** (long), **ul** (unsigned long), **ll** (long long), **ull** (unsigned long long).

Auch **Gleitkommazahlen** können direkt im Quellcode stehen:

Dazu müssen sie entweder einen Dezimalpunkt, einen Exponenten (z.B. 1e-2 oder beides enthalten.

Suffixe: Ohne ist’s 'n double, **f** für float und **l** für long double.

Zeichenliterale sind ganzzahlig und werden als Einzelzeichen innerhalb **einfacher** Anführungszeichen geschrieben: 'b'

Konstanten

Symbolische Konstanten können mittels der define-Direktive erreicht werden.

Tatsächliche Konstanten werden mit dem Schlüsselwort **const** definiert:

```
const double e = 2.7182
```

Der Unterschied sollte schon bemerkbar geworden sein: define-Konstanten sind Literale – daher gibt’s keine Typprüfung, sondern Implizite Typ-Konvertierung.

Const an den verschiedenen Stellen

Das const wird normalerweise immer **hinter** das als Konstant zu markierende Feld geschrieben. Das heißt, wenn ich einen konstanten Integer Wert haben will, schreibe ich

```
int const *number;
```

Der C-Standard erlaubt uns in diesem Falle (und nur in diesem Falle!) das const auch vor den int zu schreiben:

```
const int *number;
```

In beiden Fällen erzeugen wir einen Integer-Pointer, der selbst (also der Pointer) verändert werden darf. Nur der Wert unter dem Pointer kann nicht verändert werden.

```
int * const number;
```

Damit erzeugen wir einen Konstanten Pointer auf einen Integer. Das heißt, der Wert kann sich ändern, der Pointer auf den Wert nicht.

Const wirklich Konstant?

„Konstanten kann man nicht verändern“, sagt jeder – steht auch da oben von mir geschrieben. Aber das ist nicht die ganze Wahrheit. Wird eine Variable als const gezeichnet, beschreibt der Entwickler lediglich, er hat kein Interesse daran, diese Variable zu ändern, aber es ist dennoch möglich, dass sie sich selbst verändert. Ganz ganz böse Entwickler casten die Konstante auch einfach auf eine Nicht-Konstante, damit kann er sie wieder verändern. Daher gilt:

Das Schlüsselwort „const“ ist ein Vertrag, die Variable nicht zu ändern. Es gibt jedoch kein Schiedsgericht, dass einen möglichen Vertragsbruch verhindern könnte. Außer ein Absturz.

Aufzählungskonstanten

Aufzählungen (also Enumerationen) sind Folgen von konstanten ganzzahligen Werten.

```
enum boolean { FALSE, TRUE, keinenPlan};
```

boolean.FALSE = 0 boolean.TRUE = 1 boolean.keinenPlan = 2

Wobei man auch ganze Zahlen direkt belegen kann:

```
enum wasAnderes { Eins = 4, Zwei = 6 };
```

Verwendung

```
enum boolean { FALSE, TRUE } bool;
enum months { JAN = 1, FEB };
bool = TRUE;
enum months mon = FEB;
printf(„%d“, bool); //Ausgabe: 1
printf(„%d“, mon); //Ausgabe: 2
```

Gültigkeitsbereiche

Alle Speicherobjekte (also Variablen und Funktionen) haben einen Gültigkeitsbereich – eine Lebensdauer, welche sich mittels bestimmter Schlüsselwörter modifizieren lässt.

Dabei gilt es hauptsächlich zwischen dem Codesegment (wo das Programm selbst liegt), dem Datensegment (wo globale, externe Variablen liegen) und vor allem zwischen dem Stack und Heap zu unterscheiden.

Im Stack liegen lokale Variablen, Rücksprungadressen und Parameter einer Funktion. Sie werden pro Funktionsaufruf in einem sogenannten Stackframe innerhalb des Stacks organisiert. Ist die Funktion zu Ende, verschwindet auch der Stackframe und damit alle Infos rund um den Funktionsaufruf.

Im Heap hingegen liegen dynamische Variablen, welche detailliert auf Seite 51, „Dynamische Speicherverwaltung“ besprochen wird.

Die 4 Gültigkeitsbereiche

- Anweisungsblock (**Block Scope**)
  - Wird innerhalb eines Anweisungsblock eine Variable deklariert/definiert, ist sie auch nur innerhalb dieses Anweisungsblocks verfügbar.

```
int main(void) { int x =0; { int y = x + 5; } printf(„%d“, y); } //y ist nicht im Scope
```

  - Lokale Variablen, formale Parameter, etc.
- Funktionsprototyp (**Function Prototype Scope**)
  - Wird eine Variable innerhalb eines Funktionsprototyps (also rein in der Deklaration), dann ist diese auch nur bis zum Ende dieses Prototyps gültig.
  - z.B. VLAs
- Datei bzw. Modul (**File Scope**)
  - Speicherobjekte, die außerhalb von Funktionen deklariert werden, können ab dem Zeitpunkt der Definition bis zum Dateiende angesprochen werden.
  - Globale Variablen
- Funktion (**Function Scope**)
  - Labels (goto) sind innerhalb einer Funktion überall sichtbar.

Lebensdauer

Ein Gültigkeitsbereich bestimmt meist auch die Lebensdauer eines Speicherobjekts. Diese Lebensdauer kann automatisch sein, also entsprechend mit dem Stackframe am Ende der Funktion und seiner Gültigkeit verworfen werden. Oder aber auch eine statische Lebensdauer sein, womit ein Speicherobjekt einen festen Platz im Speicher (während der Programmlaufzeit) bekommt und immer gültig ist. Folglich müssen diese statischen Speicherobjekte auch im Datensegment und nicht im Stack gespeichert werden.

Damit kann eine Variable grundlegend im Speicher (Stack und Datensegment), Heap oder aber im Prozessorregister gehalten werden.

Modifizierer

Extern

Ein Speicherobjekt kann überall in einem Programm (welches möglicherweise aus mehreren Dateien besteht) verwendet werden. Es ist wie eine globale Variable.

Das Schlüselwort extern erlaubt uns, einen Bezug zu einem Speicherobjekt, welches an einer anderen Stelle/Datei definiert wurde, herzustellen. Der Compiler weist den Linker an, die Verweise in anderen Dateien/Bibliotheken aufzulösen.

Bei Variablen bedeutet dies also, dass keine neue Variable, sondern eine bereits definierte Variable deklariert wird, die in einer anderen Datei zu finden ist.
Es gilt auch zu beachten, mittels extern-Deklaration setzt in den Dateien nur die Adresse der Variable, wo sie definiert wurde, ein.

Bei Funktionen ist extern das Standardverhalten und bedeutet, dass die Funktion global im Programm sichtbar ist.

Intern

Ein Speicherobjekt kann nur in jeder Datei verwendet werden in der es auch definiert wurde.

Auto

Spezifizierer für lokale Variablen, welcher meistens überflüssig ist, weil auto innerhalb eines Anweisungsblocks standardmäßig gilt. Es bedeutet, die Lebensdauer ist automatisch, also nur innerhalb des Anweisungsblocks.

Zu beachten gilt, auto „versteckt“ Variablen mit gleichem Bezeichner aus einem umschließenden Block. Auch müssen auto-Variablen explizit initialisiert werden.

Register

Mit register wird der Compiler angewiesen, die Variable möglichst lange im Prozessorregister zu halten, weil der Registerzugriff enorm schneller ist, als der Arbeitsspeicher. Die tatsächliche Auswirkung hängt aber vom Compiler ab, der den register-Modifizierer einfach ignorieren könnte.

Auf eine register-Variable kann man (aufgrund der Tatsache, dass sie im Register sein *könnte*) nicht mit dem Adressoperator zugreifen.

Static

Innerhalb einer Funktion:

Verwendung für Variable, um den Wert der lokalen Variable nach der Rückkehr aus dieser Funktion zu erhalten. Die Variable kann aber dennoch nur lokal innerhalb der Funktion angesprochen werden.

Außerhalb von Funktionen vor Variablen oder Funktionen:

Weil für static Funktionen/Variablen kein Linkersymbol erzeugt wird, ist die Sichtbarkeit auf die aktuelle Datei beschränkt. Damit lassen sich Funktionen mit demselben Bezeichner in verschiedenen Quelldateien definieren.

Reservierte Schlüsselwörter

auto	**break**	**case**	**char**	**const**	**continue**
default	**do**	**double**	**else**	**enum**	**extern**
float	**for**	**goto**	**if**	**int**	**long**
register	**return**	**short**	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	**union**	**unsigned**	**void**
volatile	**while**				
restrict	**inline**	**_Bool**	**_Complex**	**_Imaginary**	

Die Roten existieren erst in C99.

Operatoren

Es gibt Arithmetische, Logische und Vergleichsoperatoren.

Unterscheidungen

Weiteres gilt es, zwischen Anzahl der Operanden zu unterscheiden

- Unäre Operatoren, besitzen einen Operanden
- Binäre, besitzen zwei
- Ternäre, drei

Abhängig von der Position:

- Präfixform, wo der Operator vor dem Operanden steht
- Postfix, wo er danach steht
- Infix, wo der Operator zwischen seinen Operanden (a + b) steht

Abhängig von der Auswertereihenfolge

- Linksassoziativ, Auswertung von links nach rechts
- Rechtsassoziativ, von rechts nach links

Arithmetische sind +, -, *, /, sowie der Modulo Operator %, der den Rest einer Division zurückgibt. Sie sind linksassoziativ.

Logische Operatoren sind **&&**, also logisches Und, d.h. beide müssen true sein, **||** für logisches Oder (inklusive!), also eines der beiden oder beide true – und natürlich die Negation mit **!** – sie sind linksassoziativ

Vergleichsoperatoren sollten eh klar sein, es gibt **==** für gleich, **!=** für ungleich, **<** & **<=** für kleiner (gleich) und halt **>** & **>=** - sie sind linksassoziativ

Natürlich gäb‘s noch den Zuweisungsoperator =, der is rechtsassoziativ.

Wichtig: Weil Ansi-C (C89) und unser c99 keinen Booleschen Datentyp hat, müssen wir die Header-Datei stdbool.h einbinden, um mit booleschen Werten True, False zu arbeiten. In jedem Falle jedoch repräsentiert 0 == False und alles andere == True

Verkürzte Schreibweise

Die arithmetischen Operatoren lassen sich kürzer schreiben, wenn das einer der Operanden auch die Variable ist, der der Wert zugewiesen werden soll:

Operation	Bezeichnung	entspricht
Op1 += Op2	Additionszuweisung	Op1 = Op1 + Op2
Op1 -= Op2	Subtraktionszuweisung	Op1 = Op1 - Op2
Op1 *= Op2	Multiplikationszuweisung	Op1 = Op1 * Op2
Op1 /= Op2	Divisionszuweisung	Op1 = Op1 / Op2
Op1 %= Op2	Modulozuweisung	Op1 = Op1 % Op2

Overflow

Bei einem arithmetischen Überlauf gibt's ein undefiniertes Verhalten, wobei bei Zahlen ohne Vorzeichen wieder bei 0 beginnt.

Inkrement- Dekrement Operatoren

Die sind Unär, und entsprechen einer Addition/Subtraktion von 1

Operator	Benennung	Beispiel	Erklärung
++	Präinkrement	++a	a wird vor seiner weiteren Verwendung um 1 erhöht
++	Postinkrement	a++	a wird nach seiner weiteren Verwendung um 1 erhöht
--	Prädekrement	--b	b wird vor seiner weiteren Verwendung um 1 erniedrigt
--	Postdekrement	b--	b wird nach seiner weiteren Verwendung um 1 erniedrigt

Short-Circuit-Evaluation

Bei logischen Operationen kann die Auswertung eines Ausdrucks abgebrochen werden, sobald der endgültige Wert dieses Ausdrucks feststeht:

```
True || (False && True)
```

(False && True) wird nicht ausgewertet, weil True oder ... immer true ist.
Sollte es ausgewertet werden, dann wird auch der zweite Teil des zweiten Ausdrucks nicht zur Gänze ausgewertet, weil (False && ...) immer zu False evaluiert.

Präzedenzen

Operator (höchste bis niedrigste Priorität)	Assoziativität
(), [], {}, ->, ., ++, --	links
!, ~, ++, --, +, -, *, &, sizeof	rechts
(type)	rechts
*, /, %	links
+, -	links
<<, >>	links
<, <=, >, >=	links
==, !=	links
&	links
^	links
|	links
&&	links
||	links
?:	rechts
=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	rechts
,	links

Aber: A * B + C * D wird von links nach rechts bearbeitet (linksassoziativ), der Compiler kann aber selbst bestimmen, welcher Operator für + (also A * B oder C * D) zuerst ausgewertet wird. Daher nie auf eine bestimmte Reihenfolge verlassen, sondern immer explizit fordern.

Bitoperationen

Damit erlaubt man einen Zugriff auf die binäre Darstellung von Ganzzahlen.

Es gibt **&**, **|**, **^** und **~**, sowie zwei Shift-Operatoren **<<** (**Linksshift**, Bits werden nach rechts verschoben, d.h. die niederwertigsten Bits gehen verloren) und **>>** (**Rechtsshift**, also Verschiebung nach links, wo die höherwertigen Bits verloren gehen).

Auch Zuweisungsoperatoren sind für Bitoperationen da: &=, |=, ^=, <<=und >>=

Verzweigungen

If erklär i da jz net haha

If(true) führt den Anweisungsblock drunter aus, sonst net.

Wichtig ist, alles was innerhalb von { } steht, ist syntaktisch genau eine Anweisung. Das heißt für dich, du kannst, wenn nur eine Anweisung hast, die Klammern weglassen:

```
If(true)
        return „cool“
```

Switch-Case

Ermöglicht das übersichtlichere Verzweigen bei mehreren Bedinungen. Bitte niemals If mit mehr als 5 ifs machen...

```
switch(valueToCompare){
        case 1: //Value equals 1
        case „hi“: //Value equals „hi“
        default: //Wenn nix gefunden
}
```

Wenn kein „break“ vorhanden, werden die cases darunter ausgeführt („fall through“) bis zum nächsten Break oder Ende.

Bei Switch können keine Ausdrücke ausgewertet werden.

Ternärer Bedingungsoperator

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

Wenn a > b == true, dann wird max = a, sonst max = b ausgeführt. Ermöglicht ein „Just-In-Place“-Einfügen von Bedingung.

Er ist rechtsassoziativ und der einzige Ternäre Operator.

Schleifen

While

Führ die Bedingung aus, solange sie true ist. Daher muss im Körper unbedingt die Anweisung irgendwann verändert werden.

Tipp: Manchmal sind Endlosschleifen sehr praktisch und guter Programmierstil, wenn in der Schleife selbst eine Bedingung zum Abbruch führt.

For

Zuerst wird eine Laufvariable initialisiert, im Ausdruck zwei wird die Schleifenbedingung geprüft, danach wird der dritte Ausdruck ausgeführt. Vor jedem Ausführen des Körpers wird die Schleifenbedingung geprüft.

```
for(int x = 0; x < 10; x++);
```

Das Deklarieren der Laufvariable ist erst mit dem C99-Standard möglich geworden.

Es dürfen auch mehrere Anweisungen mit dem Komma-Operator getrennt verwendet werden:

```
for( int x = 0, y = 10; x < y; x++, y--);
```

Auch gilt, dass jeder der drei Ausdrücke ausfallen darf.

```
for(;;);
```

Mit Break können beide Schleifen abgebrochen werden.
Mit continue können sie einen einzelnen Schleifendurchlauf überspringen.

Wann while, wann for?

Selber entscheiden.

For, wenn Start- und Endwert bekannt. While, wenn nur Endkriterium gegeben.

Do-While

Entspricht der While-Schleife, wobei der Anweisungsblock zumindest einmal ausgeführt werden muss. Beachte, dass am Ende ein Semikolon steht!

Endlosschleifen

Sind nicht immer schlecht! Definition: Es gibt kein Abbruchkriterium. Wird z.B. auf Multitasking-Systemen oder auf Servern verwendet und warten auf eine Aktion (Polling). Sie können mittels break oder return oder gar dem exit beendet werden.

Goto

Ermöglicht einen Sprung an eine Sprungmarke. Ist in sehr wenigen Ausnahmefällen sinnvoll. In sehr wenigen. Wirklich wenigen. Wirklich. Extrem wenigen.

Typumwandlung / Typecasts

Kann implizit oder explizit erfolgen. Implizit erfolgt automatisch vom Compiler.

Explizit erfolgt durch explizite Angabe.

Implizit

Erfolgt nach bestimmten Regeln, zwischen zwei verträglichen Typen. Dabei wird nie mit etwas Kleinerem als int gerechnet, dann unsigned int.

Bei **Umwandlung von unsigned in breiteren signed**, bleibt der **Wert erhalten**.

Von **signed in gleich breiten oder breiteren unsigned**, wird **Vorzeichenbit nach vorne verlängert** und **Bitdarstellung erhalten**.

Umwandlung **von unsigned in gleich breiten signed**: Bitmuster wird im signed abgespeichert, was passiert, wenn der unsigned Typ größer ist, wird **nicht** vorgeschrieben.

Von signed- bzw. unsigned in kleinerern signed oder unsigned, hängt das Ergebnis vom Compiler ab.

Gleitkomma / Ganzzahl

Von Gleitkomma nach Ganzzahl, werden die Nachkommastellen abgeschnitten, wenn der ganzzahlige Anteil größer als der Wertebereich des Ganzzahltyps ist, ist das Verhalten undefiniert.

Ganzzahl nach Gleitkommazahl erfolgt wie erwartet. **Aber**: Liegt der Wert der Ganzzalh im Wertebereich der Gleitkommazahl, ist aber nicht darstellbar, wird der nächsthöhere/-niedrigere darstellbare Wert angezeigt.

Float, Double, long Double

Ist der Zahlenbereich des Zieltyps gleich groß oder größer als der Ausgangstyp, so ist alles kein Problem.

Wenn er jedoch kleiner ist (double -> float), liegt undefiniertes Verhalten vor.

Übliche Arithmetische Umwandlungen

Werden bei binären Operatoren durchgeführt, mit dem Ziel, einen gemeinsamen Typ der Operanden zu erhalten, der auch der Typ des Ergebnisses ist.

Vertikale Umwandlung: Wenn einer der Operanden einen höheren Datentyp hat, wird der andere zunächst in diesen umgewandelt. Wird bei Bedarf durchgeführt.

Horizontale Umwandlung: Wird **immer** durchgeführt. Zum Beispiel char in int.

Explizit

Ein Datentyp der in Klammern vor dem Ausdruck steht, folgt in einem expliziten Typecast. Der mögliche Verlust von Informationen wird damit in Kauf genommen.

Typedef

Typedef ist kein Typecast! Aber ich wollt keine Überschrift dafür einführen:

Typedef ist eine Vereinbarung auf einen neuen Typnamen:

```
typedef unsigned long un64;
```

Damit kann ich nun „un64 variableName = 0;“ machen, um einen unsigned long zu erzeugen. Achtung: Hier erfolgt keine Textersetzung (wie bei define)!

Äußerst praktisch für die Strukturen, die unten noch kommen werden.

```
typedef struct person { char name[10]; } user;
user variable = { „Hansi“ };
```

Sequenzpunkte

Sie dienen rein der C-Sprache und dem Compiler zur Auswertung der Syntax und Umwandlung in Maschinencode.

An einem solchen Punkt müssen alle bisherigen „Seiteneffekte“, also Modifikationen eines Objekts, ausgewertet sein. Davor wird nicht fortgefahren. Zwischen zwei Sequenzpunkten darf nur ein Schreibzugriff auf ein Objekt erfolgen.

Bisher bekannt sind uns das Semikolon, das Komma, die logischen Operatoren && und || und der Ternäre Operator. Auch Funktionsaufrufe und das Ende vollständiger Ausdrücke sind Sequenzpunkte.

Funktionen

Grundlegend werden Teile eines Programms unter eigenem Namen zusammengefasst. Damit lassen sich „Aufgaben“ mit einem Namen aufrufen.

Main ist die Hauptfunktion, die vom System aufgerufen wird, wenn das Programm gestartet werden soll.

Die Aufgabe einer Funktion besteht darin, aus Eingabedaten Ausgabedaten zu erzeugen. Bei **Pure-Functions** werden dabei keine Teile außerhalb der Funktion „geladen“, verändert, oder gespeichert. Es werden nur die Eingabeparameter verwendet und bei gleichem, mehrmaligen Aufruf mit derselben Parameterliste, kommt immer dasselbe Ergebnis zurück. Auch keine Pointer werden verändert.

```
[Spezifizierer] Rückgabetyp Funktionsname (Typ name, ...);
```

Funktionsname muss eindeutig sein. Spezifizierer ist optional (Speicherklasse).

Sollte eine Funktion keine Parameter erlauben, muss im der Parameterliste void stehen. Ansonsten werden Parameter erlaubt, aber nicht verwendet.

Variadische Funktionen haben eine variable Anzahl von Parametern.

Formaler und Aktueller Parameter

Formale Parameter (Parameter) werden als lokale Variablen mit dem entsprechenden Wert initialisiert (**call-by-value**)

Aktuelle Parameter (Argumente) kann ein beliebiger Ausdruck sein, bei dem eine Zuweisung stattfindet.

Bei Argumenten werden implizite Typecasts durchgeführt, dabei wird float immer zu double umgewandelt.

Erwähnenswert ist auch, dass die Reihenfolge von Argument-Auswertungen nicht definiert ist. D.h. Der Compiler entscheidet selbst, welchen Parameter er zuerst auswertet.

Deklaration von Funktionen

Da der Compiler die Konsistenz zwischen Funktionskopf und Aufruf überprüft, muss die Funktion dem Compiler beim Aufruf bekannt/deklariert sein. Weil das

aber schwer wäre, immer die Funktion vor dem Aufruf zu definieren, gibt's die **Vorwärtsdeklaration** (auch Funktionsprototyp genannt).

Dabei gilt es zu beachten, dass die Signatur nur den Rückgabetyp, den Bezeichner und die Parametertypen beachtet, nicht aber die formalen Parameternamen. D.h. dem Compiler ist's wurscht, ob da

```
int keinenPlan (int, char);          oder
int keinenPlan (int name, char lmao);      steht
```

Genau diese Prototypen/Signaturen stehen in Header-Dateien. Damit wird durch das Einbinden einer Header-Datei alle enthaltenen Funktionsprototypen bekannt.

Gültigkeitsbereiche

In einer Datei gibt es vier Gültigkeitsbereiche:

1. Datei
2. Funktion
3. Block
4. Funktionsprototyp

Namen, die in Blöcken eingeführt werden, verlieren am Ende des Blocks die Bedeutung.

Namen der formalen Paramter von Funktionen gelten nur innerhalb der entsprechenden Funktion.

Externe Variablen sind ab ihrer Deklaration bis zum Ende der Datei gültig.

Globale Variablen können in allen Funktionen einer Datei verwendet werden, gibt es eine lokale mit demselben Namen, wird dieser mehr Priorität zugewiesen. Bitte niemals globale Variablen definieren. Sie werden bei Definition automatisch initialisiert, im Gegensatz zu lokalen Variablen.

Argumentliste

Sollte grundsätzlich kein Problem sein, aber es gibt ein cooles und nützliches Feature in C: Variable Argumentlisten

In C dürfen nämlich Funktionen mit einer variabel langen Argumentliste aufgerufen werden, dazu benötigt man allerdings die Headerdatei stdarg.h, genauer folgende Datentypen und Makros:

Typ/Makro	**Syntax**	**Beschreibung**
va_list	va_list argPtr;	Abstrakter Datentyp (Argumentzeiger), mit dem die Liste der Parameter definiert wird und der Zugriff realisiert
va_start	void va_start(va_list argPtr, lastarg)	Argumentliste initialisiert den Argumentzeiger argPtr mit der Position des ersten optionalen Arguments. lastarg muss der letzte fixe (vor den Optionalen) Parameter in der Liste übergeben werden
va_arg	type va_arg(va_list argPtr, type);	Gibt das optionale Argument zurück, auf das argPtr zurzeit verweist und setzt den Zeiger auf das nächste Argument. Mit type gibt man den Typ des zu lesenden Arguments an
va_end	Void va_end(va_list argPtr);	Hiermit kann man den Argumentzeiger argPtr beenden

Ein Beispiel dazu auf Seite 48 (+1 o. 2 seiten)

Allgemein wichtige/Standard Funktionen

main

Dieee Main-Funktion. Der Startpunkt jedes Programmes. Ihr darf (und kann/sollte/will) man Parameter übergeben. Bzw. genaugenommen kann das System beim Programmstart einige Parameter übergeben. Und der Programmierer ist dazu angehalten, diese Parameter auch zu lesen und darauf einzugehen. Dazu müssen wir unser aber mit den Parametern erst auseinandersetzen:

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

argc (Argument Counter, Typ int)

Wenn dieser größer als 1 ist, dann enthalten argv[1] bis argv[argc-1] die Programmparameter.

Argv (Argument Vector, Typ char *[] bzw. char **)

Dieser Parameter ist ein Array von Zeigern auf char, wobei der erste Zeiger meist auf den Programmnamen zeigt (aber nicht immer).

Damit wir mit diesen Zeichenketten was anfangen können, müssen wir sie durch entsprechende Funktionen umwandeln: **strtol, strtoul, strtod**.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Konvertiert den Anfangsteil von `nptr` in einen Long-Wert zur Basis `base`.

Wenn `endptr != NULL`, dann enthält `endptr` die Adresse des erste Zeichens, das nicht konvertiert werden konnte.

- Wenn keine Zeichen vorhanden waren, dann enthält `endptr` den Wert von `nptr` und `strtol` gibt 0 zurück.

man-Seiten lesen!

PrintF

```
printf(„Whasuppppp“);
```

printf ist in der stdio.h enthalten und ermöglicht eine Ausgabe an der Konsole.

Escape Sequences

Manche Zeichen können aufgrund ihrer Charakteristik nicht einfach dargestellt werden, z.B.: Tabulatoren, Zeilenumbrüche. Dafür brauchen wir Escape Sequences, also spezielle Zeichen, die genau diesen Wert repräsentieren.

Escape Sequences sind immer nur ein einzelnes Char, auch wenn sie optisch aus zwei bestehen:

```
\t    Insert a tab in the text at this point.
\b    Insert a backspace in the text at this point.
\n    Insert a newline in the text at this point.
\r    Insert a carriage return in the text at this point.
\f    Insert a formfeed in the text at this point.
\'    Insert a single quote character in the text at this point.
\"    Insert a double quote character in the text at this point.
\\    Insert a backslash character in the text at this point.
```

printf Format Identifiers/Strings

Variablen können in Printf mittels den Format Identifiers ausgeben werden:

```
Printf(„That is a value: %d“, value);
```

Dabei wird die Parameterliste verwendet um die Placeholder zu ersetzen und entsprechend den Typen zu formatieren:

Zeichen	Bedeutung
d,i	Ausgabe als ganze (positive oder negative) Dezimalzahl (Datentyp `int`).
u	Ausgabe als ganze positive Dezimalzahl (Datentyp `unsigned int`).
o	Ausgabe als ganze positive Oktalzahl (Datentyp `unsigned int`).
x, X	Ausgabe als ganze positive Hexadezimalzahl in Klein- bzw. Großschreibung (Datentyp `unsigned int`).
c	Ausgabe eines einzelnen Zeichens (Datentyp `int`).
s	Ausgabe einer Zeichenkette (wird noch ausführlich besprochen).
f	Ausgabe als Fließkommazahl (Datentyp `double`).
e, E	Ausgabe mit Exponentialdarstellung mit kleingeschriebenem „e“ bzw. großgeschriebenem „E“.
g, G	Ausgabe als Exponentialzahl bzw. als Dezimalzahl in Abhängigkeit vom Wert.
a, A (C 99)	Ausgabe als Exponentialzahl in hexadezimaler Darstellung.
p	Ausgabe als Adresse (Ausgabe ist implementierungsabhängig).
n	Speichert in einer als Argument angegebenen Variable des Typs `int` die Anzahl der bisher ausgegebenen Zeichen.
%	Ausgabe des %-Zeichens.

Placeholder-Modifikationen

Nun können wir die Placeholder auch weiter formatieren, um unsere Ausgabe auch allen Wünschen Folge leisten zu können:

Die Platzhalter haben grundsätzlich folgende Struktur:

```
%[flags][weite][.genauigkeit][modifizierer]typ
```

Flags:

Zum Beispiel:

- „-“ für linksbündige Ausgabe oder „+“ um positive Zahl mit Vorzeichen auszugeben
- 0 um führende Nullen aufzufüllen, # für alternative Form
- Leerzeichen (wird vorangestellt, wenn erstes Zeichen kein Vorzeichen)

Weite:

Gibt die Mindestanzahl der Zeichen bei der Ausgabe an. Ist die Ausgabe kürzer, wird mit Leerzeichen aufgefüllt, ist sie länger, wird sie ignoriert.

Genauigkeit:

Anzahl der Nachkommastellen, bzw. Stellen für Ganzzahlen. Wenn nicht vorhanden, wird bei Gleitkommazahlen mit 6 Nachkommastellen gearbeitet.

Modifizierer:

Lässt den Wert auf den Typ in der nachfolgenden Tabelle umwandeln

Typ:

Siehe obige Tabelle.

Zeichen	Bedeutung
h	Ganzzahlumwandlung als short (signed, unsigned)
hh	Ganzzahlumwandlung entspricht char (signed, unsigned)
j	Ganzzahlumwandlung entspricht einem Argument vom Typ `intmax_t` oder `uintmax_t`.
l	Ganzzahlumwandlung als long(signed, unsigned)
ll	Ganzzahlumwandlung als long long(signed, unsigned)
L	Eine folgende a-, A-, e-, E-, f-, F-, g- oder G-Umwandlung entspricht einem long double-Argument
t	Ganzzahlumwandlung entspricht einem Argument vom Typ `ptrdiff_t` (Differenz zweier Zeiger)
z	Ganzzahlumwandlung entspricht einem Argument vom Typ `size_t` oder `ssize_t`.

getchar & putchar

Liest ein einzelnes Zeichen von der Tastatur, liefert einen Integer zurück.

Dabei gilt zu beachten, dass getchar() immer nur ein Zeichen pro Aufruf liefert, nicht aber nur ein Zeichen einlesen lässt. Das heißt „abc“ liefert ‚a‘, ‚b‘, ‚c‘ ‚EOF‘.

Putchar hingegen gibt ein Zeichen auf der Konsole aus, auch hier muss ein Integer übergeben werden.

```
while ((c = getchar()) != EOF)
putchar(c);
```

scanf

Damit lässt sich zeichenweise eine Folge von Eingabefeldern einlesen. Für jedes Eingabefeld muss eine Adresse vorhanden, deren Datentyp mit dem Eingabefeld übereinstimmen muss. Die Adresse (Pointer) bekommt man mit **&**x.

Formatierung ist ähnlich zu printf(), aber es gibt spezielle Anforderungen – dazu die Manpage (man scanf) lesen.

Um verbleibende Zeichen aus dem Puffer zu lesen bietet sich folgendes Snippet an:

```
while (scanf("%d %d %d", &a, &b, &c) != 3) {
    while (getchar() != '\n');
} …
```

Alternativ ist die Verwendung von **fgets**() sehr ratsam!

sizeof

Dient zur Ermittlung der Größe von Datentypen und Objekten im Hauptspeicher. Kann eine Variable oder einen Datentyp als Parameter übernehmen. Bei Übergabe einer Variable wird der Typ der Variable genommen.
Ausdrücke werden nicht evaluiert, nur bestimmt.

Präzedenz ist unglaublich hoch:

```
sizeof x + y == sizeof(x) + y
```

Die Rückgabe ist unsigned long oder unsigned int, und architekturabhängig, vom Typ size_t (in stddef.h)

Rekursion

Wenn eine Funktion, Datenstruktur oder whatever sich selbst definiert, nennt man dies Rekursion.

Funktionen sind dann **rekursiv**, welche die Funktion selbst direkt oder indirekt aufruft.

Iterativ sind Funktionen, wenn bestimmte Abschnitte mehrfach durchlaufen werden (iteriert werden).

Rekursion und Iteration sind äquivalent, wenn man sie zueinander umformen kann. Dabei gilt zu beachten, dass rekursive Lösungen meistens wesentlich kürzer sind, aber schnell viel Speicher beanspruchen. Hier tritt der Stack das erste Mal in den Vordergrund:

Stack

Im Stack wird das Programm, bzw. genauer die derzeit aktive Funktion (und darunter die aufrufenden Funktionen) zwischengespeichert. Nach Beendigung einer Funktion wird diese vom Stack gelöscht und die Funktion, welche diese zuvor aufgerufen hat, kann mit ihrem vorherigen Zustand weitergeführt werden. (Call-Stack).

Mit jedem Funktionsaufruf wird ein Stack-Frame angelegt, in dem formale Parameter, lokale Variablen, Rücksprungadresse, etc. gespeichert werden. Diese werden mit Beendigung wieder freigegeben (darum sind lokale Variablen danach nicht mehr verfügbar.

Jetzt ist auch klar, warum Rekursion zu einem Stack-Overflow führen kann.

Nested-Functions

Sind zwar im C-Standard nicht definiert, moderne Compiler unterstützen sie jedoch. Wir verwenden in der Vorlesung und den Prüfungen keine!

Pointer (Zeiger)

Jede Speicherzelle besitzt eine Nummer (Adresse). Ein Pointer ist eine Variable, welche eine solche Adresse abspeichert. D.h. unter der Adresse des Pointers befindet sich eine Adresse zu einer anderen Variable.

Dabei gilt es zu beachten, dass Pointer und die Speicherobjekte, auf welche der Pointer zeigt, vom Typ her gekoppelt sind. Damit kann der Compiler Typkompatibilität garantieren und Zeiger-Arithmetik erlauben.

Auf 32-Bit Systemen ist ein Pointer in der Regel 32-Bit lang, auf 64-Bit Rechnern normalerweise 64-Bit lang.

Warum?

Weil man Speicherbereiche dynamisch reservieren, verwalten & löschen kann.

Mit Zeigern kann man Datenobjekte direkt (call-by-reference) an Funktionen übergeben, Funktionen selbst als Argumente an andere Funktionen übergeben und Rekursive Datenstrukturen wie Listen und Arrays erstellen.

Definitionen (* und &)

Grundsätzlich wird ein Pointer wie folgt deklariert:

```
int *pointer;
int* pointer;
```

Beide Notationen sind erlaubt, die letztere sei die moderne Variante. In beiden Fällen wird nur Speicher für die Darstellung der Adresse reserviert.

Vor der Initialisierung hat ein Pointer keinen definierten Wert, er zeigt auf eine zufällige Speicherstelle. Man kann ihn allerdings mit NULL initialisieren, damit zeigt der Pointer auf 0x0 (NULL ist definiert in stddef.h) und lässt sich auch auf valide Daten prüfen. Daher einen nicht sofort verwendeten Zeiger immer mit NULL initialisieren.

Dereferenzierungsoperator *

Er dereferenziert (wer hät's gedacht) die Adresse und liefert damit das Objekt hinter der Adresse.

```
int value = *pointer; //value nimmt den Wert in der Adresse, die in pointer gespeichert liegt an
```

Adressoperator &

Mithilfe diesem lässt sich die Adresse einer Variable auslesen. Das heißt,

```
int* pointer = &variable;
```

**&alpha ist äquivalent zu alpha,*
wenn alpha eine Wert-Variable sei.

Zeiger und Funktionsargumente

Wird in C eine Variable einer Funktion übergeben, wird diese **immer** als Kopie an den formalen Parameter übergeben – **call-by-value**.

Übergibt man nun einen Zeiger, wird die Adresse, die der Zeiger hält, kopiert und an die Funktion übergeben. Damit lässt sich allerdings dennoch der Wert unter dem Zeiger verändern und ist damit auch außerhalb der Funktion sichtbar. Das nennt sich **call-by-reference**.

Sollte jedoch der Zeiger selbst verändert werden, wirkt sich dies nicht auf den ursprünglichen (kopierten) Zeiger aus.

---

*Arrays und Pointer sind **nicht** dasselbe.*

Ein Array belegt automatisch einen Speicherbereich, der nicht verschoben oder in der Größe verändert werden kann.
Einem Zeiger muss man einen Wert zuweisen, damit er auf einen belegten Speicher zeigt und darf auch an eine Adresse zeigen, die kein Anfang eines Speicherblocks ist.

Der Arrayname, also die Arrayvariable selbst, zeigt immer auf die Adresse des ersten Elements. Sie kann also auch einem Pointer zugewiesen werden.

```
alpha[0] == *alpha
alpha[i] == *(alpha + i) //Verweis auf Zeigerarithmetik!
```

Das gilt auch umgekehrt. Der Compiler arbeitet selbst nur mit Zeigern.

```
array == &array[0]
```

Zeigerarithmetik

Dafür stehen uns folgende Operatoren zur Verfügung:

+, -, +=, -=, ++, --

Die Addition und Subtraktion funktioniert gleichermaßen – d.h. Addiere ich 2 zu einem Pointer hinzu, wird 2x die Länge des Typs des Zeigers addiert.

```
int* pointer = 0x0010; //Annahme, Int = 4 Bytes groß, also sizeof(int) == 4
pointer + 2 == pointer + sizeof(int) + sizeof(int) == 0x0010 + 4 + 4 == 0x0018
```

Die Subtraktion erfolgt gleichermaßen, aber halt mit Minus.

Rechne ich einen Pointer minus einem Pointer, werden die Anzahl der Elemente zwischen den zwei Zeigern zurückgegeben. Das kann unter Umständen zu einem Fehler führen. Sollte daher nur innerhalb des gleichen Arrays durchgeführt werden.

Natürlich lassen sich auch Vergleichsoperatoren anwenden, wobei hier immer die Speicheradressen verglichen werden. Dazu müssen jedoch beide Pointer denselben Typen besitzen.

Mehrdimensionale Zeigerarithmetik

Für die Mehrdimensionalen Arrays verweise ich höflichst auf Seite 38 ☺

Hier geht's nur ums Rechnen damit.

Sei folgender Array gegeben:

```
int arr[4][3];
```

dann ist arr die Adresse der ersten Zeile, arr + 2 die Adresse der dritten Zeile. Das heißt, beim Addieren zu einem Mehrdimensionalen Array wird immer die Größe des Typs (int = 4 Bytes) mal der Anzahl an Unterelemente (hier 3) addiert:

0x0010 = arr, dann ist arr + 2 = arr + (2 * (sizeof(int) * 3)) = arr + 2*12 = 0x0034

```
arr + 2 == &(arr + 2)
&(arr + 2) + 1 == Addresse des zweiten Elements in der dritten Zeile!
&(arr + 2) + 1 == &(&(arr + 2) + 1)
```

Arrayzeiger

(Seite 33) Arrayzeiger != Zeigerarrays (Seite 39)

Ein Arrayzeiger ist ein Zeiger auf einen Array, der explizit als Zeiger gewünscht ist.

```
int (*ptr)[10];
```

Ist ein Zeiger ptr, der auf ein Array mit 10 Integern zeigt. **Ohne die Klammern wärs ein Array mit 10 Integer-Zeigern!**

Beispiel:

```
int arr[4][2] = { {1,2}, {3,4}, {5,6}, {7,8} };
int (*pz) [2];
pz = arr;
printf("%d %d %d %d %d", pz[0][0], *pz[0], **pz, pz[2][1], *(*(pz + 2) + 1));
```

Zeiger auf Zeiger

Jop, ein Zeiger der auf einen Zeiger zeigt, der auf ein Objekt zeigt.

```
char **textpointer; //äquivalent zu:
char *textpointer[];
```

Das heißt im Zeiger auf den Zeiger steht die Adresse des Pointers, in dem die Adresse des Objekts steht.

Bei dynamisch erzeugten, mehrdimensionalen Arrays ist das nahezu unabdinglich.

Zeiger auf Void

Sollte der Typ der Variable, auf den der Pointer zeigt, noch nicht bekannt sein, darf man ihm den Typ void zuweisen. Solche Zeiger können jedoch nicht dereferenziert werden, dafür aber zu einem späteren Zeitpunkt in einen Typen umgewandelt werden.

Solche Pointer sind generische Pointer, der zu allen anderen Typen kompatibel ist. Daraus resultiert auch eine **nicht gewährleistete Typprüfung**!

Zeiger auf Funktionen

In C ist ein Funktionsname eine Adresskonstante. Ähnlich wie bei Arrays.

Durch Dereferenzieren kann man dann die Funktion aufrufen. Alternativ könnte man den Zeiger auf die Funktion als Parameter an andere Funktionen übergeben.

```
int (*fptr)(const char*,...);
fptr = printf;
(*fptr)("Hello World"); //äquivalent zu printf("Hello World");
```

Restriktionen bei Zuweisungen

```
int n = 5;
double x;
int *p1 = &n;
double *pd = &x;
x = n;
pd = p1;          // Fehler !
```

```
int *pt;
int (*pa)[3];
int arr1[2][3];
int arr2[3][2];
int **p2;
…
pt = &arr1[0][0];
pt = arr1[0];
pt = arr1;        // Fehler !
pa = arr1;
pa = arr2;        // Fehler !
p2 = &pt;
*p2 = arr2[0];
p2 = arr2;        // Fehler !
```

Wichtige Bemerkungen

Typ-Qualifizierer

Const

Mit dem Schlüsselwort „const“ lässt sich ein Zeiger konstant definieren. Dabei gilt es allerdings einige „unterschiedliche Konstanten“:

- Konstanter Zeiger:
  - Const steht rechts vom Stern,
  - Damit ist der Pointer selbst konstant, er zeigt immer auf dieselbe Adresse, der dereferenzierte Wert darf sich ändern!

```
int * const c_iptr1;
```

- Zeiger auf konstante Daten
  - Const steht links vom Stern, manchmal auch vor dem Datentyp
  - Zeiger darf verändert werden, die Daten dahinter aber nicht.
  - Konstante Parameter für Funktionen, die Funktion darf die Parameter also nicht überschreiben (z.B.: printf(const char*, ...)

```
Int const *c_iptr2;
```

- Konstante Zeiger auf konstante Daten
  - Const steht links und rechts vom Stern
  - Kombination aus beidem

Volatile

Volatile-Variablen werden vor jedem Zugriff neu aus dem Hauptspeicher eingelesen. Das heißt, der Compiler wird nix optimieren/cachen.

Restrict

Damit wird angegeben, dass ein Zugriff **ausnahmslos nur** über diesen restrict-Zeiger erfolgen darf. Es darf kein anderer Zeiger darauf zeigen.

Während der Compiler das zwar optimieren kann, liegt die Verantwortung beim Programmierer.

Arrays (Vektoren)

Arrays sind die Zusammenfassung von mehreren Variablen gleichen Typs.

```
Typname Arrayname [GROESSE]
```

Größe muss eine positive ganze Zahl sein und definiert, wieviel Speicher geordert wird. Heißt, ist der Array Typ ein Integer (Annahme: 4-Byte), mit der Größe 10, wird 10x 4-Byte = 40-Byte reserviert.

Danach kann jedes Element mit einem Index erreicht werden:

```
counter[4] = 3; //Setzt das 5te Element (Im Speicher 4x 4-Bytes vom Anfang aus losgerechnet) auf 3
```

Achtung: Beim Überschreiten/Unterschreiten des zulässigen Index-Bereiches werden **keine** Übersetzungsfehler erzeugt. Das heißt, es greift auf irgendwelche Speicherbereiche vor bzw. nach dem Array zu, wo sich andere Daten befinden. Das

kann zu einem Absturz oder Fehlerhafte Daten führen. Das Verhalten ist nicht definiert.

Werden Arrays bereits bei der Definition initialisiert, ist die Größenangabe obligatorisch:

```
float farr2[] = {0.75, 1.0, 2.5, 5.5, 6.0};
```

Sollten jedoch nicht alle Elemente angegeben werden, muss die Größe dennoch dastehen. Die restlichen Werte werden mit 0 initialisiert:

```
Int iarr[5] = {100, 200};
```

Auch bestimmte Elemente können ausgewählt werden (C99):

```
int iarr3[5] = {100, [4] = 500};
```

Lokale Arrays können vollständig mit 0 initialisiert werden:

```
int iarr2[5] = {0};
```

Arrays als Funktionsparameter

Es wird nicht das Array übergeben (also kopiert), sondern nur die Anfangsadresse (also der Pointer) des Arrays. Daher ins **alle Änderungen an einem Array** in einer Funktion ausnahmslos **auch außerhalb der Funktion sichtbar**. Will man dies verhindern, kann man const verwenden. Dann sind alle Elemente eines Arrays als Konstanten definiert.

Daher sind folgende Funktionsköpfe gleichbedeutenden:

```
void function(int arr[]);
void function(int arr[9]); //die neun wird vom Compiler ignoriert und hat nur für den Programmierer einen „beschreibenden“ Wert
void function(int* arr);
void function int *arr);
```

Aufrufmöglichkeiten:

```
int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int *iptr;
iptr = arr;
 ...
funktion(arr); //Ident
funktion(iptr ); //Ident
funktion(&arr [2]); //Hier wird die Adresse zum 3ten Element übergeben
```

Mehrdimensionale Arrays bei Funktionen

```
void  f(int (*p)[4]){ }
void  f(int p[][4]){ }
```

Diese beiden Funktionen sind äquivalent! In den leeren Klammern dürfen auch Zahlen stehen.

Bei den folgenden Klammern jedoch **muss** eine Zahl mitangegeben werden, weil die Größe des Arrays bekannt sein muss!

Wichtige Bemerkungen

Zusammengesetzte Literale (Compound Literals)

Seit C99 können Literale verwendet werden, um Arrays darzustellen.

```
(int[]){4, 5, 6}
```

Man „castet“ also die Liste in ein Array (nicht wirklich, aber man merkt sichs so vllt). Diese Arrays haben keine Bezeichner.

Vergleiche

Niemals mit == Operator vergleichen, weil man dann nur die Pointer miteinander vergleicht, nicht den Inhalt.

Also entweder mit Schleife alle Elemente einzeln prüfen oder mit der speziellen Funktion memcmp().

Länge ermitteln

Sizeof auf ein Array gibt die Größe in Bytes des gesamten Arrays zurück. Daher sollte man nochmal durch die Größe des jeweiligen Datentyps dividieren, um die Länge des Arrays zu erhalten.

Wichtig, wird ein Array an eine Funktion übergeben, ist dort nur der Pointer des Arrays bekannt. Sizeof darauf gibt die Größe des Pointers, nicht des Arrays zurück!

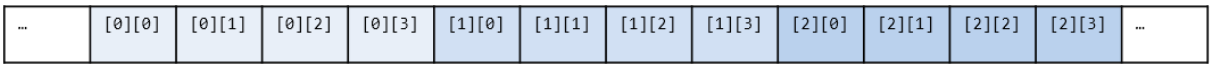
Mehrdimensionale Arrays

```
int matrix[2][4]; //Mehrdimensionaler Array mit 2 Zeilen und 4 Spalten
```

Funktioniert eh überall gleich wie eindimensionale. Jede Dimension ist im Grunde nur ein Array im Array. Das ist auch im Standard so festgeschrieben, weil C nur eindimensionale Arrays kennt. Aber **jedes Element darf beliebigen Typs** (auch Arrays) sein.

Interessant ist die Speicherbelegung. Den Mehrdimensionale Arrays werden in C zeilenweise sequentiell hintereinander gespeichert:

```
int arr[3][4];
```



Bei der Übergabe von mehrdimensionalen Arrays kann die Größe der ersten Dimension weggelassen werden, alle weiteren müssen jedoch angeführt werden:

```
void multiply_matrices(int a[][N], int b[][P], int c[][P]) { … }
```

Werden diese Dimensionen über Variablen spezifiziert, müssen sie in der Parameterliste vorangestellt werden:

```
int sum(int rows, int columns, int arr[rows][columns])
```

Zeigerarrays

(Seite 33) Arrayzeiger != Zeigerarrays (Seite 39)

Auch Zeigerarrays sind möglich:

```
int* counter[10]; // Array mit 10 Integer-Zeigern
```

Solche Zeigerarrays werden oft als Alternative für zweidimensionale Arrays verwendet:

```
char numbers[10][50] = { "eins", "zwei", "drei", "vier", "fünf", "sechs", "sieben", "acht" ,
"neun ", "zehn" };
char *numbers2[10] = { "eins", "zwei", "drei", "vier", "fünf", "sechs", "sieben", "acht ",
"neun ", "zehn" };
```

Für den ersten Array werden jedoch 500 Bytes, für den zweiten lediglich für jeden String der dafür benötigte Speicherplatz reserviert.

Strings

Strings sind im Grunde nur char-Arrays (**oder** Zeiger vom Typ char). Dabei wird das Ende des Arrays mit einem Null-Zeichen \0 (Bytes, das nur 0en enthält) markiert. Strings sind daher immer um 1 Zeichen größer als gedacht ;)

Der formale Parameter einer Funktion, der als String übergeben wird, kann daher auch char[] sein.

Es bietet sich jedoch an, die Header-Datei string.h einzubinden, um viele Funktionen auf Strings anwenden zu können (strlen, strcpy, strcmp, ...)

Fehlt das Null-Zeichen, wird das nächste gesucht → manchmal Absturz. Es sollte niemals die Größe eines char-Arrays angegeben werden, der Compiler errechnet die Länge immer selbst!

Funktionsargument String

Der Formale Parameter einer Funktion, welcher einen String erwartet, kann vom Typ char* oder char[] sein.

Strukturen

Strukthuren sind Ansammlungen von mehreren Variablen, welche unter einem einzigen Namen gesammelt werden. Die versch. Variablen können auch verschiedene Typen (und sogar wiederrum Strukturen) sein. Damit lassen sich also eigene Typen zum Organisieren von Daten (Zusammenfassen von Daten die thematisch zusammengehören, z.B. alle Eigenschaften eines Autos).

Deklaration und Definition

Pretty straightforward:

```
struct point {
        int x;
        int y;
}

struct point pt;
```

```
//Alternativ „Inline-Definition“:
struct point{
        int x; int y;
} pt;

//oder auch „Anonymous-Struct“:
struct {
        int x; int y;
} pt;
```

Wobei die rechten neet so elegant sind, weil die Anonymous schwierig zum „Weitergeben“ an eine Funktion ist. Die Inline-Definition kann man schon verwenden, aber klarer ists mit dem linken Beispiel – und Readability ist das A und O eines guten Programmierers.

Initialisierung

Grundsätzlich initialisiert man einfach mit einer Liste als Zuweisung

```
struct point maxpt = { 300, 200 };
struct point maxpt = { .y = 200, .x = 300 } //Seit C99 möglich!
```

Dabei gilt, nicht alle Member müssen initialisiert werden (am Ende). Diese Werte haben einen undefinierten Wert!

Statische und Globale Structs werden mit 0 initialisiert, wenn nicht anders angegeben.

Eine Struktur darf auch einer anderen Struktur desselben Typs zugewiesen werden.

Zugriff

Um nun auf die einzelnen Member zugreifen zu können, verwenden wir den „.“-Operator.

```
pt.x + pt.y
```

Weil die structs zumeist als Pointer vorliegen (Funktionsparameter übernimmt meist einen Pointer, weil sonst ja call-by-value eine Kopie des structs erzeugt), hat man in C ein sehr praktisches Feature eingebaut:

Der „->“-Operator erlaubt direkten Zugriff auf die Member eines Structs, wenn das Struct als Pointer vorliegt.

```
struct point *pt;
pt->x; //ist äquivalent zu (*pt).x
```

Grundlagen zu Structs

Zeiger und Funktionen auf Structs

Zeiger auf Strukturen funktionieren gleich wie Zeiger auf Variablen und Zeiger auf Arrays (wobei man nicht mit [i] indizieren kann, wär aber auch unlogisch).

Auf Member eines Struct-Pointers können wir einfach mit dem „->“-Operator zugreifen.

```
struct person p;
p->name; //statt (*p).name
```

Funktionsparameter

Wir können entweder einzelne Member eines Structs wie herkömmliche Variablen übergeben, oder aber das gesamte Struct als Kopie (call-by-value) oder aber eine Referenz/Pointer auf dasselbe Struct (call-by-reference).

Dazu einfach entsprechend die Funktion deklarieren:

```
void whateva(struct point pt);
```

```
void whateva(struct point *pt);
```

Beachte die Methoden haben verschiedene Signaturen!

Weil bei letzteren Variante ein Zeiger übergeben wird, können die Inhalte des Structs verändert werden. Sei dies nicht gewünscht, können wir auch das Schlüsselwort „const“ verwenden. Dabei gilt es aber zu beachten, dieses Const ist nur ein Versprechen von der Methode, die Inhalte nicht zu verändern! Beziehungsweiße der Entwickler hatte kein Interesse daran, das struct zu ändern, es ist aber definitiv möglich.

An dieser Stelle bitte unbedingt nochmal Seite 15 „Konstanten“.

Bei der Übergabe via call-by-value, wird das gesamte Struct kopiert, verbraucht daher viel Speicherplatz und Zeit. Die Rückgabe sollte eigentlich selbsterklärend sein, wird einfach ebenfalls äquivalent verwendet.

Struct-Arrays

Joah, wenn des ah Erklärung braucht, bist ah Trottel.

```
struct person personList[10] = { { „Hansi Hinterseer“, „Irgendwo 10, 1000 Wien“ },
{ „Emil die Ente“, „Entenhausen 12“} };
struct person trump = { „Donald Trump“, „White House :^“};
personList[2] = trump;
```

Struct-Literal

Wir können Structs auch als Literal für Listen verwenden, allerdings erst mit C99. Dabei wird eigentlich eine Liste gecastet:

```
(struct person){ „Mir fallen keine neuen Namen mehr ein“, „Technikerstraße 1“ }
```

Offset

Das „offsetof“-Makro (definiert in stddef.h) erlaubt uns das ermitteln des Abstands eines bestimmten Members von der Anfangsadresse des Structs ausgehend.

```
offsetof(Person, alter); //Gibt das Offset zu alter in Person an.
```

Union

Unions sind sehr ähnlich wie Strukturen (syntaktisch äquivalent), haben aber einen anderen (für mich unnötigen) Verwendungszweck.

Defacto kann ein Union mehrere Variablen deklariert bekommen, der Speicher wird entsprechend dem Größten der enthaltenen Variablen reserviert (also nicht für jede Variable, nur für die Größte) und tatsächlich liegt immer nur ein einziger Wert in dem Union, entsprechend einem der definierten Datentypen seiner Variablen.

Okay, wenn des nicht kapiert hast, hier ein Beispiel:

```
typedef union foo{
        int ival;
        float fval;
} bar;
bar var;
var.ival = 12; //Absofort können wir auf var.ival zugreifen
```

```
var.fval = 5.5f; //Nun auf var.fval, aber nicht mehr auf var.ival
var.ival = 6; //Jetzt wieder auf var.ival
```

Das heißt, immer nur ein Member der Variable ist wirklich aktiv („vereint“/“unioned“) durch die union. Damit sparen wir Speicher, müssen aber Bedenken, dass die Member keinen Zusammenhang haben sollten.

Es darf immer nur das erste Element der Union bei der Initialisierung angegeben werden / bzw. mit C99 auch durch den Membername (wie structs).

Bitfelder

Bitfelder können einzelne Member einer Struktur oder Union sein. Sie bestehen aus ganzzahligen Variablen, die wiederrum aus einer bestimmten Anzahl von Bits bestehen, mit denen man einzeln (wie bei Strukturelementen) innerhalb eines Bytes arbeiten kann.

Sind vor allem für embedded Systems (wegen geringem Speicherplatz) und Zugriff auf Hardware wichtig.

Der Typ muss ein ganzzahliger Typ sein, die Breite die Anzahl an Bits

```
Typ Elementname : Breite
```

```
typedef struct zeit {
  unsigned int stunde;
  unsigned int minute;
  unsigned int tag;
  unsigned int monat;
  unsigned int jahr;
  unsigned int sommerzeit;
} Zeit_t;
```

```
typedef struct zeit {
  unsigned int stunde:5;
  unsigned int minute:6;
  unsigned int tag:5;
  unsigned int monat:4;
  unsigned int jahr:11;
  unsigned int sommerzeit:1;
} Zeit_t;
```

Im obigen Beispiel sind die Elemente stunde, minute, ... alles Bitfelder. Das heißt, wir können wie bei Structs darauf zugreifen, werden aber als Bitfelder intern verwaltet:

```
Zeit_t z1 =;
z1.stunde = 23; //z1->stunde == 23
```

Während im linken Beispiel für jeden Member 32 Bit (beispielsweise) verwendet werden == 192 Bits, werden rechts insgesamt nur 32 Bit verwendet

Einschränkungen

Bitfelder sind nicht adressierbar! Das heißt, kein sizeof erlaubt, kein Adressoperator, kein scanf(!), kein Array von Bitfelder-Elementen, kein offsetof-Makro

Die Anordnung von Bitfeldern ist undefiniert und dem Compiler überlassen (bzw. der Architektur!) Zumeist ist der Zugriff auf diese Bitfeldelemente langsamer, als der herkömmliche Datentyp dazu.

Listen

Aus der Vorlesung „Praktische Informatik“ sollte bekannt sein, was verkette Listen sind. Sie sind eine Folge von Elemente, die dynamisch während der Laufzeit verlängert, bzw. verkürzt werden. Dabei ist nicht garantiert, dass die Elemente im

Speicher hintereinander liegen, sondern es wird ein Member der Struktur als Verweis auf das nächste genutzt.

C unterstützt Listen nicht standardmäßig, das heißt, wir müssen uns selbst welche basteln. Dazu gibt es vier Varianten, welche alle in der Library queue.h zufinden sind (Infos auch unter „man 3 queue“)

Einfach verkette Liste

Jedes Element hält einen Wert und einen Pointer auf das nächste Element der Liste.

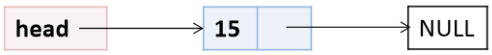
```
struct node {
        int value;
        struct node *next
};
typedef struct node node_t;
typedef struct node *nodeptr_t;
nodeptr_t head = NULL;
```

Das wäre die einfachste Variante, dabei können wir nun über node_t bzw. nodeptr_t einfach unsere Liste als Struktur verwenden. Der Anfang einer Liste wäre über den Pointer head erreichbar.

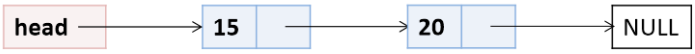
Einfügen

Zum Einfügen brauchen wir nur den head auf ein neu erzeugtes Element zeigen lassen und den nodeptr innerhalb des elements auf null. Sollte schon ein Element existieren, suchen wir das letzte Element in der Liste und lassen den nodeptr des letzten Elements auf das neu Erzeugte zeigen:

Einfügen von 15



Einfügen von 20 (am Ende der Liste)

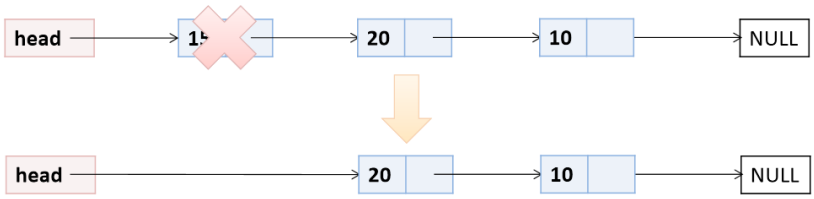


```c
void insert_node(nodeptr_t new) {
    if (head == NULL) {
        head = new;
        new->next = NULL;
    } else {
        nodeptr_t help = head;
        while (help->next != NULL) {
            help = help->next;
        }
        help->next = new;
        new->next = NULL;
    }
}
```

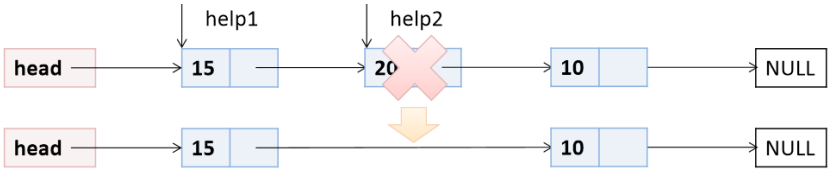
Löschen

Um ein Element zu löschen, müssen wir grundlegend zwei Fälle unterscheiden:

Wird der erste Knoten in der Liste gelöscht, brauchen wir einen Hilfszeiger (Um den Zeiger innerhalb des ersten Knoten nach dem Löschen als head zu verwenden, davor brauchen wir aber head um den Knoten zu löschen:



Sollte jedoch ein beliebiger anderer Knoten gelöscht werden, brauchen wir sogar zwei Hilfszeiger (Weil wir zusätzlich wissen müssen, wo wir uns in der Liste befinden)



```
void delete_node(int val) {
     if (head != NULL) {
          if (head->value == val) {
               nodeptr_t help = head->next;
               free(head);
               head = help;
          } else {
               nodeptr_t help1 = head;
               while (help1->next != NULL) {
                    nodeptr_t help2 = help1->next;
                    if (help2->value == val) {
                         help1->next = help2->next;
                         free(help2);
                         break;
                    }
                    help1 = help2;
               }
          }
     }
}
```

Doppelt verkettete Liste

Dabei wird jedem Element noch ein Zeiger auf das vorherige Element hinzugefügt, wodurch wir vor und rückwärts durch die Liste navigieren können.

```
struct elem {
    char name[20];
    struct elem *next;
    struct elem *prev;
};
```

Listen mit Container

Die Liste wird über einen Listencontainer angesprochen, was eine Überprüfung vereinfacht und uns erlaubt, zusätzliche Informationen über die Liste zu speichern.

```c
/* Listenelement */
struct ielem {
	int val;
	struct ielem *next;
};

/* Listencontainer */
struct ilist {
	int count;
	struct ielem *first;
};
```

Nachfolgend kommt ein komplettes Modul, wobei dies exakt von den Vorlesungsfolien kopiert wurde:

Header-Datei

```c
#ifndef _ILIST_H
#define _ILIST_H

/* List element */
struct ielem {
	int val;
	struct ielem *next;
};

/* List container */
struct ilist {
	int count; // size(first)
	struct ielem *first;
};

/* Initializes the new list */
struct ilist *ilist_init(void);

/* Adds a new element at position pos or at the end of the list */
void insert_node(struct ilist *list, int pos, int val);

/* Adds a variable number of elements at the end of the list */
void add_nodes(struct ilist *list, int num, ...);

/* Removes an element from position pos and does not change anything if the
 * element is not found in the list*/
void delete_node(struct ilist *list, int pos);

/* Frees the memory allocated by the list */
void ilist_free(struct ilist *list);

/* Prints the content of the list */
void list_nodes(struct ilist *list);

#endif
```

Implementierung

```
struct ilist *ilist_init(void) {
    struct ilist *new = malloc(sizeof(struct ilist));
    if (new == NULL) {
        fprintf(stderr, "out of memory");
        exit(EXIT_FAILURE);
    }
    new->count = 0;
    new->first = NULL;
    return new;
}

void insert_node(struct ilist *list, int pos, int val) {
     struct ielem *tmp = list->first, *last = NULL, *new;
     int i = 0;

     new = malloc(sizeof(struct ielem));
     if (new == NULL) {
          fprintf(stderr, "out of memory");
          exit(EXIT_FAILURE);
     }
     new->val = val;
     new->next = NULL;

     while (tmp && i++ < pos) {
          last = tmp;
          tmp = tmp->next;
     }
     new->next = tmp;
     if (last)
          last->next = new;
     else
          list->first = new;
     list->count++;
}

void add_nodes(struct ilist *list, int num, ...) {
   va_list zeiger;
   va_start(zeiger,num);
   while (num-- > 0)
      insert_node(list, list->count, va_arg(zeiger,int));
   va_end(zeiger);
}
```

„...“ = Variable Argumentliste**, Seite 26**

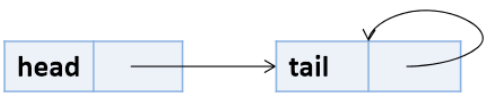
```c
void delete_node(struct ilist *list, int pos) {
    struct ielem *tmp = list->first, *last = NULL;
    int i = 0;
    while (tmp && i++ < pos) {
        last = tmp;
        tmp = tmp->next;
    }
    if (!tmp)
        return;
    if (last)
        last->next = tmp->next;
    else
        list->first = list->first->next;
    free(tmp);
    list->count--;
}

void list_nodes(struct ilist *list) {
    struct ielem *tmp = list->first;
    printf("size: %d [ ", list->count);
    while (tmp != NULL) {
        printf("%d ", tmp->val);
        tmp = tmp->next;
    }
    printf("]\n");
}

void ilist_free(struct ilist *list) {
    struct ielem *l = list->first, *tmp;
    while (l) {
        tmp = l;
        l = l->next;
        free(tmp);
    }
    free(list);
}
```

Erweiterung um einen tail-Zeiger

Wir erweitern unsere Implementierung um einen Zeiger auf das letzte Element, was zwar mehr Verwaltungsaufwand kostet, aber auch Vorteile bringt. Zu bemerken gilt allerdings, dass damit kein direkter Zugriff auf das letzte Element ermöglicht wird!



```c
struct list_element {
    int val;
    struct list_element *next;
};
struct list_element *head, *tail;


void list_init(void) {
    head = malloc(sizeof(struct list_element));
    tail = malloc(sizeof(struct list_element));
    if (head == NULL || tail == NULL) {
        printf(".......Out of memory\n");
        exit(1);
    }
    head->next = tail->next = tail;
}
```

```c
void insert_node(int elem) {
     struct list_element *new_element = malloc(sizeof(struct list_element));
     if (new_element == NULL) {
          printf(".......Out of memory!\n");
          exit(1);
     }
     new_element->val = elem;
     struct list_element *list_ptr = head;
     while (list_ptr->next != list_ptr->next->next) {
          list_ptr = list_ptr->next;
     }
     new_element->next = list_ptr->next;
     list_ptr->next = new_element;
}
```

Vor- und Nachteile verketteter Listen

Vorteile

Einfügen und Entfernen ist weniger speicheraufwändig als bei Arrays.

Die Anzahl der Elemente und der dafür nötige Speicherplatz muss nicht im Voraus bekannt sein, sind also leicht erweiterbar.

Nachteile

Kein Direktzugriff auf ein beliebiges Element, sondern man muss alle Elemente durchlaufen bis zum gewünschten.

Zeiger belegen zusätzlichen Speicherplatz.

Dynamische Speicherverwaltung

Meistens wissen wir Programmierer nicht, was der Nutzer eigentlich will. Daher müssen wir ihm manchmal die Möglichkeit geben, unser Programm dynamisch zu verwenden, z.B.: Indem wir ihm die Möglichkeit geben, beliebig viele Zahlen einzugeben.

Mit dem bisherigen Wissen wird das schwierig (nicht unmöglich, aber hässlich), daher brauchen wir die dynamische Speicherverwaltung. Diese wird erst wirklich durch das Einbinden der stdlib.h möglich. Dazu später, erstmal Theorie:

Heap

Für die gesamte dynamische Speicherverwaltung kommt der Speicher vom Heap. Dieser kann dynamisch zugewiesen werden und ist bis zum expliziten Wunsch des Programms, ihn freizugeben, über Zeiger verfügbar. Verlieren wir den Zeiger, entstehen Speicherlecks! Auf der anderen Seite geht die Lebensdauer der dynamisch erzeugten Variablen über die Gültigkeit des umschließenden Blocks hinaus.

Heap fährt nicht nach dem LIFO-Prinzip, weil ja alles dynamisch sein muss – damit können aber auch Probleme entstehen, wie z.B.: Kein Speicherplatz mehr, obwohl theoretisch noch genügend da ist, aber nur einzelne kleine Blöcke.

Heap-Fragmentierung

Dieses oben genannte Problem nennt man Heap-Fragmentierung. Bei Speicheranforderung wird ein zusammenhängender Block angefordert. Passiert dies oft, können freie Blöcke entstehen, die zu klein für eine neue Speicheranforderung sind entstehen – der Heap wird fragmentiert. Damit kann die Summe der Lücken viel größer sein, als die Speicheranforderung.

Um diesem Missstand entgegen zu wirken, sollte man dynamische Speicherverwaltung wirklich nur verwenden, wenn es sinnvoll ist, und dann auch sehr sparsam („Pooling“, einen Großen Block anfordern und Teile davon verwenden). In manchem Situationen macht es auch Sinn, *alloca()* zu verwenden. Dabei wird der Speicher vom Stack, nicht vom Heap angefordert. **Achtung**, damit kommen aber auch alle Eigenschaften des Stacks mit, das heißt, am Ende der Funktion ist der dynamische Speicher futsch. Weiteres war dies nie im ANSI-Standard so vorgesehen!

Speicherlecks

Was sind Speicherlecks?

Stell dir vor, du gehst zum McDonalds. Du tretest vor dem Verkäufer und bestellst einen Hamburger. Du zahlst, nimmst ihn... aber du isst ihn nicht. Du sagst zum Verkäufer, bitte noch einen Hamburger. Du zahlst ihn, du nimmst ihn... aber du isst ihn nicht. Du sagst zum Verkäufer, bitte noch einen Hamburger. Du zahlst ihn, du nimmst ich... aber du isst ihn nicht.

Das machst du immer und immer wieder.

Irgendwann wird es dem Verkäufer zu viel, er bringt dich um und nimmt sich alle Hamburger wieder.

DAS sind Speicherlecks, daher unheimlich gefährlich, weil du mit deinen Speicherlecks (Hamburger-Käufe) dem Verkäufer/Prozessor an irgend einem Punkt dermaßen auf den Sack gehst, dass er dich einfach tötet und den Speicher/Hamburger für die nächsten Kunden/Programme zurückholt.

Aber wie entstehen Sie?

Stell dir vor, du warst der lästige Kunde im McDonalds. Nach jedem Kauf eines Burgers hast du ihn auf den Boden gestellt, bist aufgestanden und vergessen, dass der Burger am Boden steht. Defacto gehört der Burger dir, du könntest ihn essen, aber du findest ihn nicht mehr.

Beim Programmieren ist das gleich, du bekommst einen Speicher (bzw. die Adresse auf den Anfang des Speichers). Wenn du diesen Zeiger nun verlierst/überschreibst, kannst du nicht mehr darauf zugreifen, aber der Speicher gehört trotzdem dir. Du hast ein Speicherleck erzeugt.

Speicheranforderung

Die Alloc-Befehle

Gaaaaaanz ganz wichtig: wir übergeben den Allokationsbefehlen immer eine Größe. Diese Größe entspricht einer Anzahl an Bytes, die reserviert werden. Hier treten häufig Laufzeitfehler auf, die lange unbemerkt bleiben!!

Wenn wir die Größe übergeben, müssen wir überprüfen, welche Größe wir übergeben:

```
double *nums = malloc( maxNumbers * sizeof(nums)); //Falsch
double *nums = malloc( maxNumbers * sizeof(*nums)); //Richtig
```

Beim ersten übergeben wir die Größe des Pointers, nicht des Doubles!

malloc

```
void *malloc(size_t size)
```

size_t ist ein eigens definierter vorzeichenloser Ganzzahlentyp.

Beim Aufruf wird ein zusammenhängender Speicherblock von ***size* Bytes** am Heap reserviert. malloc liefert die Anfangsadresse zu diesem Speicherbereich als void-Pointer zurück. Der Compiler betreibt implicit Typecasting. Liefert malloc NULL, schlug die Speicherallokation fehl!

calloc

```
void *calloc(size_t nmeb, size_t size)
```

Hier wird die Anzahl und die Größe als Parameter übergeben. Das Verhalten ist grundsätzlich gleich wie malloc. Es werden jedoch (***nmeb*size***) **Bytes** am Heap reserviert.

Wichtig: **calloc initialisiert** im Gegensatz zu malloc den reservierten Speicher mit **0**! Folglich braucht calloc aber auch mehr Zeit.

realloc

```
void *realloc(void *ptr, size_t size)
```

Mit dieser Funktion können wir den bereits reservierten Speicherplatz (über ptr) während der Laufzeit auf ***size* Bytes** anpassen. Dabei kann der zurückgelieferte Pointer/Speicherblock an einer anderen Stelle stehen, als der vorherige, der Inhalt von ptr bleibt aber erhalten! Das impliziert, dass es möglicherweise notwendig ist, den Inhalt von ptr zu kopieren (rechenaufwändig).

Beachte, realloc(NULL, size) == malloc(size)!
realloc(ptr, 0) == free(ptr)
Wird der Speicherbereich verkleinert, wird der verkleinerte Teil freigegeben, der Inhalt davor bleibt unverändert.
Zeigt noch ein Zeiger auf die alte Adresse (vor realloc mit neuer adresse), ist diese Adresse ungültig!

Wenn realloc scheitert, bleibt der Speicherblock unter ptr unverändert!

free

Wir müssen nicht mehr benötigten Speicher **immer freigeben**. Ohne Ausnahme. Verlass dich nicht darauf, dass der Speicherplatz bei Programmende freigegeben wird!

```
void free(void *ptr)
```

Der Speicher unter ptr **muss mit malloc, calloc oder realloc reserviert** worden sein, darf **nur einmal ausgeführt** werden und setzt den **Zeiger nicht auf NULL**.

Sollte nach free über den Zeiger auf den Speicher zugegriffen werden, ist das Verhalten undefiniert. Genauso, wenn der übergebene ptr ein falscher Pointer ist!

Wenn die Speicherallokation fehlschlägt

Dann nicht in Panik geraten! Speicheranforderung reduzieren, alternativen in Erwägung ziehen und möglicherweise die Anforderung aufteilen. Niemals das Programm beenden, das will der Nutzer nie!!!

Vielleicht Daten auch auf Festplatte zwischenspeichern und nur benötigten Speicher anfordern oder Pooling betreiben.

Zweidimensionale, dynamische Arrays

Die nxm-Matrix lässt sich als Pointer auf Pointer (int**) realisieren. Dabei wird für die Zeilen ein eigener Block und dann für jede Spalte ein eigener Block reserviert. Auch beim Freigeben müssen wir gleich (aber umgekehrt) vorgehen.

```
int **init_matrix(int dim1, int dim2) {
  int **matrix;

  if ((matrix = calloc(dim1, sizeof(int *))))
  for (int i = 0; i < dim1; i++) {
    if (!(matrix[i] = calloc(dim2, sizeof(int)))) {
      printf("No memory for line %d", i);
      exit(EXIT_FAILURE);
    }
  }
  else {
    printf("Out of memory ...");
    exit(EXIT_FAILURE);
  }

  return matrix;
}

void free_matrix(int **matrix, int dim1) {
  for (int i = 0; i < dim1; i++)
    free(matrix[i]);
  free(matrix);
}
```

Zugriff auf ...	Möglichkeit 1	Möglichkeit 2	Möglichkeit 3
0. Zeile, 0. Spalte	`**matrix`	`*matrix[0]`	`matrix[0][0]`
i. Zeile, 0. Spalte	`**(matrix+i)`	`*matrix[i]`	`matrix[i][0]`
0. Zeile, i. Spalte	`*(*matrix+i)`	`*(matrix[0]+i)`	`matrix[0][i]`
i. Zeile, j. Spalte	`*(*(matrix+i)+j)`	`*(matrix[i]+j)`	`matrix[i][j]`

Analyse-Tools

Valgrind

Analyseframework für Unix-basierte Systeme zum Debuggen, Profilen und zur dynamischen Fehleranalyse in Programmen.

Wird verwendet um **Speicherlecks** zu finden, Cacheanalyse, Ressourcenbedarf, etc

Es arbeitet mit dem Binärcode der Anwendung, daher Sprachenunabhängig, aber optimiert für C/C++.

```
valgrind -tool=<NAME> ./binaryFile [ARGUMENTS]
```

<NAME> = Name des Tools (**memcheck**, cachegrind).

Memcheck

Ist das meistverwendete Tool von Valgrind. Es **erkennt fehlerhafte Zugriffe auf Speicherbereiche**, Benutzung von nicht initialisierten Variablen, **falsches Freigeben von Speicher** (z.b. doppeltes), **Speicherlecks**, Verwendung von Adresszeigern mit ungültigem Wert (**dangling pointers**).

Sehr nützlich!

Für mehr Tools lies die manpage oder google, bin ja nicht da nicht da Herold.

Modulare Programmierung

Bisher haben wir uns nur mit der Implementierung eines Programmes beschäftigt. Nun sollten wir uns auch um das Ganze rundherum Gedanken machen:

- Problemanalyse
- Systementwurf
- Programmentwurf
- Implementierung und Test
- Betrieb und Wartung

Gottseidank unterstützt C das sogenannte Structured Design, damit wir das Programm in kleinere Einheiten zerlegen können, die untereinander möglichst wenig Querbeziehungen haben müssen. Wir nennen diese Einheiten auch Funktionen.

Eine einzelne Funktion kann ein Problem lösen und die Lösung zurückgeben. Pure Functions sind Funktionen, die dabei nichts weiter als die Parameter zur Übergabe verwenden – und bei derselben Übergabe immer dieselbe Rückgabe haben.

Diese Funktionen selbst können jedoch wiederrum in Module gefasst werden, um gleichwertige und ähnliche Funktionen zusammenzufassen. Die Module wiederrum lassen sich in einer Library zusammenfassen.

Zum Beispiel:
Funktion: Add
Modul: Calculations
Library: Maths.

Modulares Design

Primär gilt es, Informationen zu kapseln. Wir bieten eine Schnittstelle und das darunterliegende bleibt verborgen, für den Anwender und Nutzer der Schnittstelle ist das sowieso uninteressant. Folglich braucht ein Modul grundsätzlich zwei Schnittstellen (Interfaces):

Die Export-Schnittstelle, über welche ein Modul Ressourcen für andere Module zur Verfügung stellt. Der Rest bleibt verborgen (Information Hiding). Somit kann der Entwickler selbst bestimmten, was von seinem Modul benutzt werden kann (Export-Interface) und was verborgen bleibt (Information Hiding).

Die Import-Schnittstelle, die ein Modul zur Implementierung ein anderes Moduls nutzt. Damit ermöglichen wir, ein „importiertes“ Modul zu ersetzen, sofern die Export-Schnittstelle dort dieselbe ist.

Die Beiden Schnittstellen werden vom Linker wie Zahnräder aneinandergelegt und wenn sie kompatibel sind zum Laufen gebracht.

Information-Hiding

Um Funktionen (seltener Globale Variablen) dem Modul vorzubehalten, also dass niemand außer das Modul darauf zugreifen darf, können wir das Schlüsselwort **static** verwenden.

Header-Datei

Die Header-Datei enthält die Funktionsprototypen und stellt damit die Export-Schnittstelle dar. Die C-Datei, welche die Header-Datei realisiert, stellt die Implementierung dar. Es wäre folglich einfach idiotisch, in der Header-Datei Information-Hiding zu betreiben, weil es die Schnittstelle ist, die anderen zur Verfügung gestellt wird...

Dennoch gehören Quelldatei und Header-Datei logisch zusammen. Das sollte sich auch im Dateinamen wiederspiegeln.

Daher gibt es einfache Regeln für Header-Dateien:

- Funktionen, welche außerhalb der Datei benutzt werden sollen, sind als Prototyp in der Header-Datei zu finden
- Dateien, die einen Aufruf einer Funktion enthält, welche nicht in der Datei definiert wurde, inkludiert die entsprechende Header-Datei
- Die Quelldatei, welche die Definition einer Funktion enthält, muss die Header-Datei ebenfalls inkludieren.

Ein- & Ausgabe

Die Ein- und Ausgabe wird über sogenannte Streams realisiert, dabei unterscheidet man zwischen Textströme und Binäre Ströme.

Ein Textstrom ist eine Folge von Zeilen, die alle mit \n enden. Dabei werden alle sichtbaren ASCII-Zeichen und einige Steuercodes verwendet.

Bei binären Strömen wird Byte für Byte gearbeitet, ohne Konvertierung.

Puffering

Weil es nicht wirklich sinnvoll wäre, immer Zeichen für Zeichen zu verarbeiten, unterscheidet man drei Arten von Puffering bei Streams:

- Vollgepuffert, wobei Zeichen erst übertragen werden, wenn der Puffer (z.B. 4096 Bytes) voll ist.
- Zeilengepuffert, wobei immer nur Zeile für Zeile übertragen wird **oder** der Puffer voll ist.
- Ungepuffert, wobei die Zeichen sofort übertragen werden.

Standard-Streams

stdin, stoud, stderr sind die Standard-Streams, welche jedes C-Programm von Anfang an besitzt. Sie sind eigentlich nur Zeiger auf ein File-Objekt.

stdin

Der Standard Input ist die Eingabe (z.B.: über Tastatur), sie wird zeilenweise gepuffert.

stdout

Der Standard Output ist gewöhnlich mit dem Bildschirm zur Ausgabe verbunden und wird ebenfalls zeilenweise gepuffert.

stderr

Die Standardfehlerausgabe (Standard Error Output) ist gewöhnlich auch mit dem Bildschirm verbunden, aber die Ausgabe erfolgt ungepuffert.

Dateien

Weil man manchmal Daten auch in Dateien schreiben will (halte ich für ein Gerücht!) wurden auch Standardfunktionen für das Öffnen von Dateien eingeführt. Dabei wird ein Speicherobjekt vom Typ FILE angelegt und bei erfolgreichem Öffnen ein Zeiger darauf zurückgegeben. Dieses FILE-Objekt hält alle nötigen Informationen zur Ein- und Ausgabe.

FILE-Typ

```
FILE *fp;
```

Damit deklarieren wir einen Pointer auf eine FILE-Struktur (in stdio.h), welche zum Arbeiten mit Streams benötigt wird. Der FILE-Pointer zeigt dabei nicht auf den Speicher, sondern eigentlich auf eine bestimmte Position im Stream, welche beim Lesen oder Schreiben verändert wird.

stdin, stdout, stderr sind automatisch geöffnete FILE-Zeiger.

Überblick der Funktionen

Öffnen

fopen

```
FILE *fopen(const char *restrict filename, const char *restrict mode);
```

filename: Spezifiziert den Dateinamen (Pfad)
mode: Spezifiziert den Zugriffsmodus, wie auf die Datei zugegriffen wird:

Mode	Beschreibung
r	Öffnen zum Lesen
w	Öffnen zum Schreiben (muss nicht existieren, wird erzeugt/überschrieben)
a	Öffnen zum Anhängen (muss nicht existieren, wird erzeugt/überschrieben)
r+	Lesen & Schreiben, startet am Anfang
w+	Lesen & Schreiben, überschreibt Datei
a+	Lesen & Schreiben, hängt am Ende an

Liefert Zeiger auf Stream zurück. Schlägt fopen fehl, liefert es NULL.

Wurde geschrieben und soll nun gelesen werden, muss unbedingt die Schreiboperation mit fflush, fsetpos, fseek, rewind abgeschlossen werden.

Wurde gelesen und soll geschrieben werden, muss der Schreibzeiger positioniert werden mit fseek, fsetpos oder rewind.

Natürlich muss der ausführende Benutzer auch die Berechtigung dazu haben, die Dinge zu tun, die unser Programm tun soll.

Soll eine Datei im Binärmodus geöffnet werden, müssen wir nur ein „b“ an das Ende des mode-Strings anhängen: rb, ab, wb, ... (Bei Linux ist das unnötig).

Bearbeiten, Manipulieren, Ändern

Formatiert:

fprintf

```
int fprintf(FILE *restrict fp, const char *restrict format, ...)
```

Arbeitet grundsätzlich wie printf, jedoch für den angegebenen Datenstrom. Der erste Parameter ist ein Zeiger auf den Strom, in den geschrieben wird und der Rückgabewert zeigt an, wie viele Zeichen tatsächlich geschrieben wurden (negativ bei Fehler).

Zum Beispiel:

```
if( fprintf(fp, „Hallo“) < 0) ...;
```

fscanf

```
int fscanf(FILE *restrict fp, const char *restrict format, ...)
```

Arbeitet grundsätzlich wie scanf, jedoch für den angegebenen Datenstrom. Der erste Parameter ist ein Zeiger auf den Strom, aus dem gelesen wird und der Rückgabewert zeigt an, wie viele Konvertierungen vorgenommen wurden. Liefert aber 0, wenn die vorgefundenen Daten nicht den geforderten Datentypen entsprechen (bzw. EOF bei Fehler oder Dateiende).

Format-Specifiers:

```
% [*] [width] [modifiers] type
```

*	Einlesen, aber nicht speichern, also einfach ignorieren
width	Weite, Feldbreite, max Number of characters
modifiers	Längenangabe, (z.b. andere größe von int bei %d)
type	Konvertierungsspezifizierer wie gewohnt.

Anstelle des Konvertierungsspezifizierers kann auch ein Suchmengenkonvertierer verwendet werden:

```
%[bezeichner] //Es wird eingelesen, bis Zeichen kommt, das nicht in der Liste bezeichner vorkommt.
%[^bezeichner] //Lesen, bis Zeichen vokrommt, das in Liste bezeichner vorkommt
```

Zeichenweise:

putc

fputc

```
int fputc(int c, FILE *fp);
int putc(int c, FILE *fp);
int putchar(int ch); //==putc(ch, stdout);
```

Schreibt ein einzelnes Zeichen in den Stream an der „Stelle“ fp.

Die Funktionen geben das geschriebene Zeichen zurück, bzw. EOF bei Fehler.

getc

fgetc

```
int fgetc(FILE *fp);
int getc(FILE *fp);
int getchar(); //gleichwertig zu getc(stdin);
```

Tun alle das gleiche, nämlich ein einzelnes Zeichen im Stream einlesen. Es liefert das einzeln gelesene Zeichen zurück, bei Fehlern liefert es EOF.

Wollen wir das eingelesene Zeichen retournieren, können wir

```
int ungetc(int c, FILE *fp);
```

Dieses Zeichen würde beim nächsten Lesen wieder gelesen werden.

Stringweise:

fputs

```
int fputs(const char *restrict str, FILE *restrict fp);
int puts(const char *str);
```

Erlaubt Zeilenweises schreiben, wobei fputs den nullterminierten String str in den Stream fp schreibt, das Terminierungszeichen aber **nicht**!

puts gibt den String str auf stdout aus und hängt das Newline-Zeichen an.

Der Rückgabewert ist nicht negativ bei Erfolg, sonst EOF.

fgets

```
char *fgets(char *restrict buf,
            int n,
            FILE *restrict fp);
```

Ermöglicht Zeilenweises einlesen, wobei vom Strom fp bis zu n-1 Zeichen in den Puffer buf eingelesen werden und am Ende das Terminierungszeichen angehängt wird.
New-Line-Zeichen werden ebenfalls mitgespeichert.

Der Lesevorgang wird foglich bei Zeilen- oder Dateiende oder nach n-1 Zeichen beendet, und retourniert einen NULL-Zeiger, wenn nichts gelesen wurde oder ein Fehler auftrat, ansonsten bekommen wir die Anfangsadresse von buf zurück.

Binär:

fwrite

```
size_t fwrite(void *restrict ptr,
              size_t size_of_elements,
              size_t number_of_elements,
              FILE *restrict fp);
```

Schreibt binäre Daten, wobei diese plattformabhängig sind!!!

ptr:	Zieladresse für die Daten,
size_of_elements:	Größe einer zu lesenden Einheit in Byte (z.B. sizeof(int), sizeof(my_struct)
number_of_elements:	Anzahl der zu lesenden Einheiten
fp:	Stream, aus dem gelesen wird
Rückgabe:	Anzahl der gelesenen Einheiten.

fread

```
size_t fread(void *restrict ptr,
             size_t size_of_elements,
             size_t number_of_elements,
             FILE *restrict fp);
```

Liest Blöcke von Daten, welche rein binär behandelt werden.

Positionieren

fseek

```
int fseek(FILE *fp, long offset, int position);
```

fp ist der zu verändernde File-Zeiger,

offset die Anzahl der Bytes, um welche die aktuelle Position des Zeigers im Strom verschoben werden soll, ausgehend von der Stelle, die durch Position angegeben wird, darf auch negativ sein.

Position kann einer von drei Werten sein:

SEEK_SET: Anfang der Datei
SEEK_CUR: Aktuelle Position des angegebenen FILE-Zeigers
SEEK_END: Ende der Datei.

Rückgabewert ist 0 bei Erfolg, EOF-Flag gelöscht; ungleich 0 bei Fehler

ftell

```
long ftell(FILE *fp);
```

Liefert die aktuelle Schreib-/Leseposition im Stream (nützlich für fseek)

fgetpos

fsetpos

```
int fgetpos(FILE *fp, fpos_t *restrict pos);
int fsetpos(FILE *fp, const fpos_t *pos);
```

Ermittelt, bzw. setzt die aktuelle Position in einem Stream (Abstand zum Anfang in Bytes), wobei pos für fsetpos von fgetpos ermittelt werden muss.

Beide geben 0 bei Erfolg zurück (EOF-Flag wird bei fsetpos gelöscht) ansonsten ungleich 0.

rewind

```
void rewind(FILE *fp);
```

Setzt die aktuelle Schreib-/Leseposition an den Anfang des Streams.

Fehlerbehandlung

Die meisten File-Operationen liefern bei Erreichen des Dateiendes sowie bei Fehlern den Wert EOF zurück.

Um nun zu unterscheiden, ob es ein Fehler oder das Dateiende ist, können wir folgende Funktionen verwenden:

ferror

```
int feof(FILE *fp);
```

Gibt ungleich 0 zurück, wenn Fehler-Flag für fp gesetzt, ansonsten 0.

feof

```
int feof(FILE *fp);
```

Gibt ungleich 0 zurück, wenn EOF Flag für fp gesetzt, ansonsten 0.

clearerr

```
void clearerr(FILE *fp);
```

Löscht das Fehler- und EOF-Flag.

Speichern/Schreiben/Puffern

setvbuf

```
int setvbuf(FILE *restrict fp, char *restrict buf, int type, size_t size);
```

type:

- _IONBF: Ungepuffert
- _IOLBF: Zeilenpufferung
- _IOFBF: Vollpufferung

buf = Puffer,
size= Größe des Puffers.

Damit können wir die Pufferung für einen Datenstrom setzen.

fflush

Um den Puffer zu leeren (und die Daten zu „speichern“ (defacto in den Stream zu schreiben)), verwenden wir fflush

```
int fflush(FILE *stream);
```

Dabei werden alle gepufferten Daten des angegebenen Ausgabestroms geschrieben. Sollte *stream* == NULL sein, werden alle geöffneten Streams geleert.

Liefert 0 bei Erfolg, sonst EOF.

Schließen

fclose

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

Schließt den zu fp gehörigen Datenstrom und leert alle damit assoziierten Puffer. Liefert 0 bei Erfolg und ungleich 0 bei Fehler.

Diese Funktion muss UNBEDINGT immer ausgeführt werden, weil sonst möglicherweise Daten verloren gehen oder ein anderes Programm während der Laufzeit nicht darauf zugreifen darf, ...

Beim Beenden werden (meist) die Datenströme automatisch geschlossen.

Dateimodifikationen

remove

```
int remove(const char *pathname);
```

Bei Erfolg wird 0 zurückgeliefert,
Bei Fehler wird -1 geliefert (Zugriffsrechte prüfen!!)

rename

```
int rename(const char *oldname, const char *newname);
```

Bei Erfolg wird 0 zurückgeliefert,
Bei Fehler wird -1 geliefert (Zugriffsrechte prüfen!!)

Errno

Einige Funktionen setzen im Fehlerfall die globale Variable errno auf einen bestimmten Wert. Damit nutzen die Funktionen die Möglichkeit, bei einer

Fehlerrückgabe von -1 in errno noch detaillierte Informationen zum Fehler (Art des Fehlers) zu hinterlegen.

fopen, fclose, remove, rename, fsetpos, fgetpos, fseek, ftell und fflush nutzen dies.

Der Wert der gesetzt wird, ist Systemabhängig und eine definierte Konstante auf die man halt dann prüfen muss – beachte, der Wert ist natürlich nur bis zum nächsten Fehler gültig ☺

Außerdem muss auf manchem Systemen errno.h eingebunden werden.

Der Fehler kann auch mit perror oder strerror ausgegeben werden, dazu aber die manpage lesen.

Ausblick auf Zukunft

Elementare E/A-Funktionen bieten noch mehr, erweiterte Möglichkeiten, sind aber nicht Bestandteil des C-Standards. Sie verwenden File-Deskriptoren (int's), um mit Verzeichnissen, Zugriffsrechten, Timestamps, etc. umzugehen. Das wird aber erst im 2ten Semester Betriebssysteme besprochen.