Zusammenfassung Programmieren Grundlagen HS 2013 C. Stucki

Umrechnungen

Hex		B		9		0		D	
Binär	MSB	1 0 1 1		1 0 0 1		0 0 0 1		1 1 0 1	LSB
Oktal	1	3		4		4		1	5

→ 4er Gruppen

→ 3er Gruppen

immer von hinten beginnen!

2^n:

$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1024 = 1k$

$2^{20} = 1M$
$2^{16} = 2^{10} \cdot 2^6 = 64k$

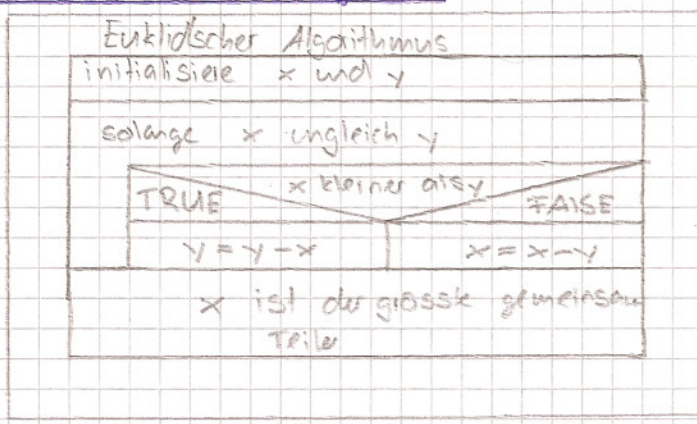
Eigenschaften von C

- Maschinennah, case-sensitive (Gross/Klein), separate Kompilierung, imperative Programmsprache, prozedurale Programmiersprache

Programmierung: Pseudocode

Dient dazu, ein Programm zu entwickeln, stich wortartig den Algorithmus beschreiben. keine grosse Formvorschrift

Nassi Schneidermann - Diagramm Buch S. 12-17



- unbedingt Rahmen um alles
- in Wörtern beschreiben, nicht mit Zeichen

Truce-Tabelle: Das Programm auf Papier durchspielen und die Werte eintragen.

Verarbeitungsschritt	x	y
initialisiere x=24 y=9	24	9
x = x-y	15	9
x = x-y	6	9
y = y-x	6	3
x = x-y	3	3
ggT ist: 3		

Datentyp legt fest:

- wie Bitmuster umgewandelt, interpretiert wird
- den Speicherbedarf
- den Wertebereich einer Variable
- die Genauigkeit einer Gleitpunktzahl

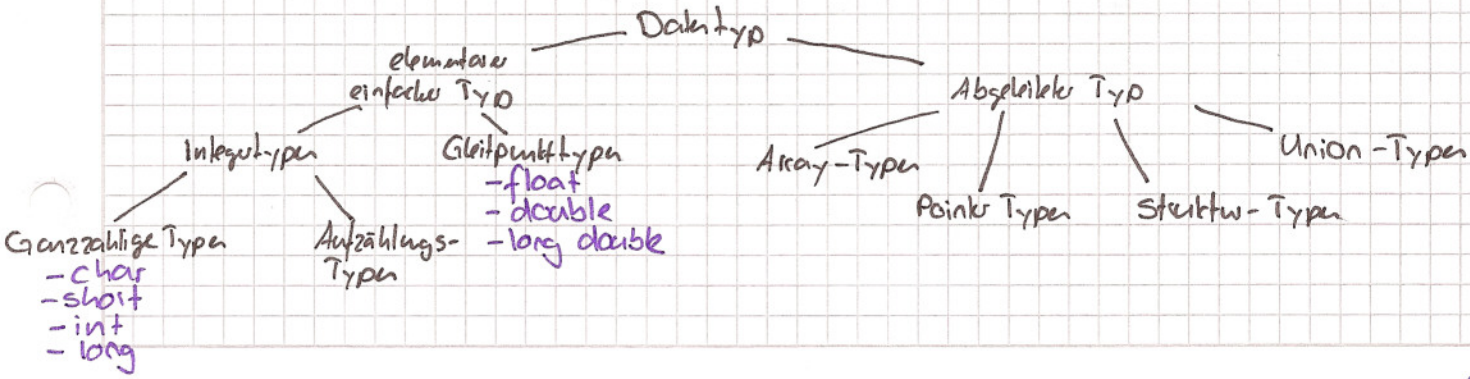
Iteration: Schleifen Buch S. 192 ff

Datentypen: Definition: legt art und Typ fest, reserviert Speicherplatz! `int a = 3;`

Deklaration: legt nur die Art und Typ fest, kein Speicherplatz! `int b;`

Definition = Deklaration + Reservierung des Speichers

Global/lokal: So lokal wie möglich, Globale werden automatisch auf 0 gesetzt
Lokale nicht!



{ { } } innerer Block | äusserer Block

Definition: Lebensdauer einer Variable Buch 9.2/S.212

Zeitspanne, ist die Zeit, in der der Compiler einer Variabeln einen Platz im Speicher gibt

- Global → während des Ganzenprogrammes
- Lokal → beim Aufruf des Blockes aktiv → beim Verlassen ungültig.
- Variabeln von äusseren Blöcken, sind innen sichtbar → umgekehrt nicht!

Funktionen unterprogramm/ Buch 216

- Gleichartige Funktionen / Programmteile zusammenfassen
- Divide et impera grosses Problem in Teilprobleme zerlegen → viel einfacher

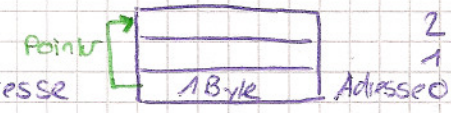
Funktionskopf: int getMiddle (int x, int y)

int → Rückgabewert; getMiddle → Funktionsname; (int x, int y) → Parameterliste

Funktionsrumpf: lokale Vereinbarungen und Anweisungen innerhalb eines Blockes

Funktionsdeklaration vor dem main()!

Pointer Buch 116



- Eine Speicherzelle ist eine Adresse
- Zeigt auf die Adresse im Speicher
- Speichergrösse unabhängig von dem Pointer-Typ. zeigt ja nur auf deren Adresse (z.B. 32Bit ≙ 4Byte)



float alpha;
float* pointer;
alpha = 1,41f;
pointer = &alpha

Referenzieren
pointer = & x
Adresse mitgeben

int-Variabel x

pointer enthält die Adresse von x

Wert mitgeben

Dereferenzieren
* pointer = 5;
heisst, das Objekt auf das der Pointer zeigt, erhält den Wert 5

Arrays Buch 124

z.B. int data[10] Array von 10 int-Typen

index von data[0] bis [9] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Sehr gut zum mehrere daten (z.B. Messungen) zu speichern!

Zeichenstring: Zeichenketten

H	A	L	L	O	\0

"Backslash 0" nicht vergessen!

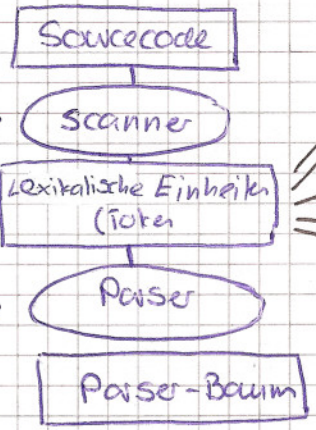
char name[15] → nur 14 zumschreiben wegen

Die Array-Grösse sollte immer mittels enum {groesse = 15}; bestimmt werden

S. 261 char[20] = "Zeichenketten" oder {"Zeichenketten"}

Lexikalische Konventionen Buch 76

Analyse des Compilers:



Erste Phase:
- Zeichengruppen finden welche nicht durch einen Trenner (Operator - Bindestrich) getrennt sind = lexikalische Einheit
- Lexikalische Einheit auf Syntax prüfen.

- reservierte Wörter (S. 81) sind immer klein
- case-sensitive alpha ≠ Alpha
  - ↳ trotzdem vermeiden

// oder /* ... */ → sind Kommentare und vereinfachen das Verständnis des Codes
→ werden vom Compiler sofort (1. Schritt) entfernt.

Namen: grundsätzlich frei wählbar (a-z, A-Z, 0-9, _,) einfach keine Zahl (0-9) an 1. Stelle

Variablen: klein beginnen und camel-case z.B. counter, maxSpeed

Funktionen: klein beginnen und camel-case z.B. init(), getSpeed()

Konstanten: Buch 82

Symbolische Konstanten	Literale Konstanten (Literale, Konstanten)
Namen, die den Wert repräsentieren z.B. #define Pi 3,141	kein Name werden durch den Wert repräsentiert z.B. 3,141

vereinfacht das abändern
- NUR GROSS und mit _ getrennt
- z.B. LIST_LENGTH mit enum

Ganzzahlige Konstanten

254	035	0x3f	-34	14L	14U	14UL
dezimal	oct	hex		long	unsigned	unsigned long

Zeichenkonstanten: ⇒ siehe S. 86

'c'	'\n'	'\x4a'	'\044'	'\\'	L'a'
		ASCII hex	ASCII oct	≙ \	Double Byte

Gleitpunktkonstanten:

254.89	-13.0	3.45e23	4.65f	3.141L
		$3.45 \cdot 10^{23}$	float konstante	long double

Aufzählkonstanten: enum { -z }; wird "durchnummeriert"

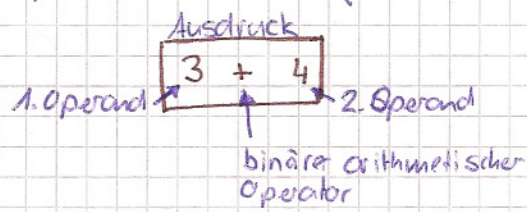
Konstante Zeichenketten: begrenzt durch " ..... " und immer mit "\0" abgeschlossen

"Claudio\0"

Operatoren & Operanden Buch 136

unär: nur einen Operanden, z.B. Minusoperator (Vorzeichenwechsel)

binär: hat 2 Operanden



Postfix- und Präfixoperatoren

Postfix-Operatoren: unäre Operationen die **nach (post)** ihrem Operanden kommen

z.B `printf("%d", i++); // i wird ausgegeben und dann inkrementiert`

Präfixoperatoren: unäre Operationen die **vor (prä)** ihrem Operanden kommen

z.B. `printf("%d", ++i); // i wird inkrementiert und dann ausgegeben.`

Anweisung und Ausdruck S.138

Ausdruck: immer Rückgabewert, kann Teil eines Grösseren Ausdruckes sein, kann zu Anweisung werden

Anweisung: nie Rückgabewert, kein Teil eines Grösseren Ausdruckes,

Selektions-, Iterations, Sprung-, Ausdrucksanweisung z.B. `while (i <= 4) { }`

Nebeneffekte:

Zu eigentlicher Aufgabe wird ein weiterer Schritt zugefügt. z.B. `j = i++;`

Aufpassen!

≙
```
j = i;
i = i+1;
```

Auswertungsreihenfolge S.141

1. Wie in Mathe, erstens Teilausdrücke in Klammern
2. Ausdrücke mit unären Operatoren, **unär ≙ von rechts nach links**
   - 2.1 zuerst postfix-Operationen auf dem Operanden
   - 2.2 dann präfix-Operationen auf dem Operanden
3. Teilausdrücke mit mehrstelligen Operationen gemäss Priorität der Operanden } S.597
4. Bei gleicher Priorität → Assoziativität (l→r; r→l)

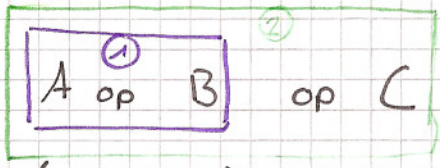
Bsp: `*p++` (= `*(p++)`) → Pointer inkrementiert, aber noch die alte Welt wird zurück gegeben, dann dereferenziert

`*++p` → Pointer inkrementiert und die neue Welt wird zurück gegeben, dann dereferenziert

`(*p)++` → zuerst dereferenziert, anschliessend der Inhalt von *p inkrementiert
→ Inhalt ausgeben und anschliessend den Inhalt inkrementieren

Assoziativität r→L ; L→r ?

- meist intuitiv richtig (manchmals L→r)
- Legt die Reihenfolge der Operatoren fest, sagt aber nichts über die Reihenfolge der Auswertung der Operatoren



- Verknüpfung hat mit Operatoren zu tun (+, −, •, :, ...)
- Auswertung mit Operanden (A, B, C, ...)

z.B. n++ − n nicht definiert!

entweder n − (n+1) = −1 falls zuerst n++ ausgewertet

oder n − n = 0 falls zuerst n ausgewertet wird

in jedem Fall n um eins inkrementiert am Schluss **AUFPASSEN!**

L-Werte & R-Werte S. 144

L-Wert (left value): wenn er sich auf ein Speicherobjekt bezieht, erfordert Schreibzugriff
kann links (und rechts) vom Zuweisungsoperator (=) stehen

R-Wert (right value): kann nur rechts stehen! kann kein Wert zugewiesen werden
• nur lese zugriff

Bsp. `i = 5·5` // ok ✓ `a·b = c` // f, (a·b) kein lvalue

`6 = a` // f, 6 kein lvalue `b − 3++` // f, 3 kein lvalue

Operatoren

- Operanden legen den Typ fest 7·5 → ganzzahl-operation
  7,0·5 → floating Point - Operation

Bedingungsoperator:

A ? B : C ≙ if (A) B; else C;

maximum ermitteln `int max = a > b ? a : b;`
a grösse b
ja: = a
nein: = b

Bitweise:

A & B	Bitweises	UND	14 & 1	≙ 1110 & 0001 = 0000
A | B	"	ODER	14 | 1	≙ 1110 | 0001 = 1111
A ^ B	"	EtOR "Addieren"	14 ^ 1	≙ 1110 ^ 0001 = 1110
~A	"	Negieren (Bitweise)		a = 9 b = ~a = [redacted] a = 1001 b = 0110
A >> B	"	verschieben	um B stellen verschieben	
		≙ : 2^n		
A << B	"	"	"	≙ · 2^n

S. 165ff **Typecast** Typumwandlung → Programmierer trägt das Risiko!

impliziete ≙ automatische Umwandlung Bsp: `int a = (int) 4.6;` `double d = double a = 4.0`

erlaubte Umwandlungen

- zwischen Skalaren grössen (Integer, Floating Points, Pointer) Daten Typen 172
- skalarer Typ ⇒ in void

Scope

- Namen einer anderen Dateien sind dem Compiler nicht bekannt.
- mit `extern ...` können sie auch von einer anderen Datei aus gelesen werden.
  ↳ kein Speicherplatz `extern int Foo_globalVariable;`
- Funktionsprototypen & Definitionen die in anderen Dateien gebraucht werden gehören ins Header-File (zB header.h) ⇒ einfügen mit `#include "header.h"`

Rekursion

Rekursion: Funktion hat Abschnitte, die sich selber aufrufen! } tauschbar ≙ umprogrammierbar
Iteration: Algorithmus mit Schleifen

Rekursiv elegant, aber ineffektiv Abbruchbedingung ist zentral!

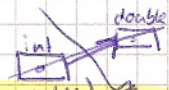
Die zwischenwerte aus Rekursion werden auf dem Stack abgelegt.
Schnelles überladen → grosse Laufzeit

Pointer 2.0 Buch 252

alpha[i] == *(alpha + i)

Arrays können nicht mit == verglichen werden, am besten mit Pointern lösbar
oder memcmp( ) bitweise vergleichen

Pointer unterschiedlicher Typen dürfen nicht einander zugewiesen werden



- void pointer können jedem zugeordnet werden z.B. `pd = (void*) pj`
  → Type cast

Addition & Subtraktion

ganze Zahlen können addiert & subtrahiert werden
→ Pointer geht dann zum nächsten Array, nicht zum Nächsten Bit
→ `n * sizeof(Typ)` Bytes! → Achtung auf Array-Bereich! nicht über Bereich hinaus!

Arrays

- Globale Arrays auf 0 initialisiert → sparsam benützen
- lokale Arrays nicht initialisiert → manuell initialisieren

`Arr[5] = {0,1,2,3,4};` keine Variabeln!
`Arr[200] = {1,3,5};` Arr[3] - Arr[199] werden = 0 gesetzt
`Arr[] = {1,2,3,4};` zählt selber die Anzahl → Arr[4]

Mehrdimensionale Arrays

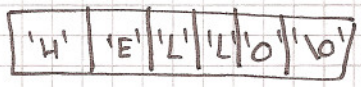
alpha[3][4] = { 0,1,2,3 }, ≙ alpha[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
{ 4,5,6,7 },
{ 8,9,10,11 };
↑ Zeilen, ↑ Spalten

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Übergabe von Arrays & Zeichenketten S. 266

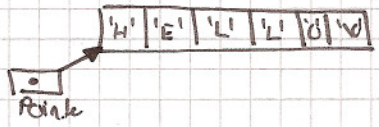
- Arraygrösse angeben
- Ausgabe mit "%s" → printf("%s\n", str);
- Übergabe einer Matrix ...(int mat[ ][5], int m, int n)
  - [ ] kann leer bleiben
  - alle anderen müssen angegeben werden
  - welches elenat

Char array:

char buf[ ] = "hello" ≙ 

buffu ≙ Anfangsadresse konstant

Pointer auf Zeichenkette:

char* pointe = "Hello"; ≙ 

pointe kann auf Adresse zeigen, ist aber nicht konstant!

Const und Pointer = lese berechtigung! S. 271

Kopieren von Zeichenketten S. 272

- Stringfunktion verwenden (evtl. selber programmieren, bei Speichermangel)
- String funktionen:
  - für Stringverarbeitung str, erkennen "\0" → siehe 275 char*
  - für Speicherverarbeitung mem, erkennen kein "\0" ⇒ Buffulänge mitgeben → 280 void*

Buffergrösse Beachten! Overflow! = Datenverlust!

Pointer auf Funktion S. 285

- jede Funktion hat ihre definierte Adresse im Speichercode
- adresse kann ermittelt werden

p = foo // ermittelt Adresse foo ≙ Funktion

ret = p(4) // = foo(4)

Strukturen S. 304

- zusammengehörige Daten zusammen lassen
- Feste Anzahl komponenten
- unterschiedliche Typen (int, char*, ...)
- jedes Feld innerhalb einer Struktur hat seinen Namen

z.B.
```
struct Name
{
  char vorname[20];
  char Nachname[20];
} ;
```
name des Structs

Strukturen können komplett übergeben werden

- Zuweisen von 2 gleichen Strukturtypen möglich
- Ermittlung der Struktur grösse mit Sizeof

Zugriff via

Normal: name.Feld

Pointer: name -> Feld

Auslesen:

struct ;

ist sehr mühsam, besser siehe typedef!

Alignment = Datentyp verlangt gerade Speicheradresse → kann also mehr Speicherplatz benötigt! mit Sizeof() herausfinden

↓ int!

Unions S.317

- ähnlich wie Struktur
- nur ein einziges Feld aktiv
- Grösse: so gross wie das grösste Feld in der Union
- mit Grossbuchstaben beginnen
- Variabeln klein beginnen
- bei Feldern kein Präfix verwenden

z.B:
```
union Name
{
    FeldTyp  feld1;
    FeldTyp  feld2;
    :
};
```

- union Name selbstdef. Typ → kann weiter verwendet werden
- Typ abhängig vom Inhalt der {}
- FeldTyp auch ein Union / Struktur möglich

Komplexere Datentypen S.336

`int* alpha[8];` ≙ alpha ist ein Array von 8 int-Pointern!

von links nach rechts lesen! Klammern haben Vorrang!

`int (*alpha)[8]`	≙	alpha ist ein Pointer auf ein Array von 8 int Werten
`int* (*beta)[8]`	≙	beta ist ein Pointer auf ein Array von 8 int Pointer
`int* function(..)`	≙	function ist eine Funktion, die ein int Pointer zurück gibt
`int (*pfunc)(..)`	≙	pfunc ist ein Pointer auf Funktion, die ein int-Wert zurück gibt
`int* (*pfunc)(..)`	≙	pfunc ist ein Pointer auf Funktion, die ein int-Pointer zurück gibt.
`char* (*delta(...))[10]`	≙	delta ist eine Funktion mit Parameterliste (...) und gibt ein Pointer auf ein Array mit 10 Pointer auf char zurück (①();②[];③*)

Typedefinitionen

Typedef ist ähnlich wie struct. Der Vorteil liegt dabei, dass nicht jedes mal beim Aufruf das Schlüsselwort geschrieben werden muss. Dies vereinfacht es stark.

Typedef sind meist Global und beginnen mit Gross Buchstaben

z.B.
```
typedef struct
{
    int  nummer;
    char name[8];
} Artikel;

int main(void)
{
    Artikel  schuh;
    Artikel* pSchuh;

    schuh.nummer = 1514;
    schuh.name = "Adidas";
}
```

→ hier wird mit Schuh eine Definition erstellt.

nur Deklaration

Portabilität: ein Typ von z.B. 16 Bits, egal auf welchem System. Dieser wird definiert und anschliessend überall zu gebrauch es muss nur die Typedif angepasst werden.
z.B. `typedef unsigned short uint16_t;`

Namensräume: S.341

Namen, die zu verschiedenen Namensräume gehören, dürfen auch innerhalb desselben gültigkeitsbereich gleich sein.
- Marken
- Namen von Struktur, Union & enums
- Jede Struktur und Union für ihre Feldnamen
- Bezeichne von Variabeln, Funktionen, typedef-Namen enum-Konstanten

Schönes Beispiel

```
typedef enum {black, red, green, yellow, blue} Color;

typedef struct
{
  int x;
  int y;
} Point;

typedef struct
{
  Point p1;
  Point p2;
  Color col;
} Line;

int main (void)
{
  Line myLine = {{12, -34}, // p1
                 {783, 12}, // p2
                 yellow}; // col
  printf ("Anfangspunkt:  (%d, %d)\n", myLine.p1.x, myLine.p1.y);
  printf ("Endpunkt:      (%d, %d)\n", myLine.p2.x, myLine.p2.y);
  printf ("Farbe:         %d\n", myLine.col);
  return 0;
}
```

Speicherklassen S. 344

- **Code** → Maschinencode — im Flash E-Prom oder RAM (nur lesen)
- **Daten** → Globale Variabeln, static Variabeln — lesen & schreiben im RAM, evtl. Register
- **Stack** → Lokale Variabeln, Parameter einer Funktion, Rücksprungadresse — lesen & schreiben im RAM, evtl. Register
- **Heap** → Dynamische Variabeln (Speicherplatz wird erst zur Laufzeit angefordert) — lesen & schreiben im RAM, evtl. Register

Für Grössere Projekte muss das Programm auf mehrere kleinere Dateien unterteilt werden können. Die Prototypen gehören dabei ins "Header-File". Möglichst kleine Schnittstellen. Hiding Infos: nur das nötige veröffentlichen!

Programm in mehreren Dateien: S. 346

Name vom Objekt

↓

Compiler → aus jeder einzelnen C-Datei wird ein eigener Maschinen-Code datei

z.B. `gcc -c datei.c`

Programm-relative Modul-Adressen ↓

Linker → Der Linker verbindet die noch offenen Referenzen. Verbindet alle Maschinen-Codes alle Dateien angeben!

↓

Adresse

z.B. `gcc -o prog.exe datei.o main.o`

Es müssen zuerst alle Dateien compiliert werden, bevor sie gelinkt werden können!

Wenn nun etwas geändert wird (z.B nur in 1 Datei) muss alles wieder eingetippt werden. Das ist mühsam und darum gibt's die **make-files**. (Auch zum Schutz, das nichts vergessen wird)

```
# makefile
cc=gcc
CFLAGS= -c -Wall
LFLAGS= -Wall
OBJ=foo.o goo.o main.o
EXE=prog.exe

${EXE}: ${OBJ}
	${cc} ${LFLAGS} -o ${@} ${OBJ}

foo.o: foo.h goo.h foo.c
	${cc} ${CFLAGS} -o ${@} foo.c

goo.o: goo.h goo.c
	${cc} ${CFLAGS} -o ${@} goo.c

main.o: foo.h main.c
	${cc} ${CFLAGS} -o ${@} main.c

clean:
	rm -f ${OBJ} ${EXE}
```

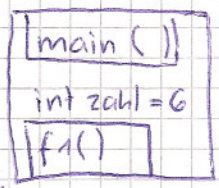
Targets: Diese können mit make angesprungen werden, z.B. make goo.o

Abhängigkeitsliste, d.h. von diesen Dateien ist das Target abhängig

Befehl, der ausgeführt werden muss, falls etwas geändert hat. Wichtig: erstes Zeichen muss ein TAB sein.

Speicherklasse extern S.347

ext1.c



ext2.c



f2() will auf zahl von ext1.c zugreifen können

// im Programm ext2.c
extern int zahl; // nun ist Sie lesbar

- kann nur in einer einziger Datei definiert werden (ohne extern)
- mit extern in anderen Dateien bekannt machen
- manuelle Definition nur bei Definition möglich
- Globale =0
- übliche weise Globale in Header-Datei
- jede C-Datei eine .h Datei!

Static Variabeln S.353

- sind im Datenbereich, nicht Stack
- Gültigkeitsbereich im Block, in dem sie definiert wird
- nur einmal vorhanden. Halten den wert, auch wenn Funktion geschlossen, beim nächsten Aufruf geht's mit dem alten wert weiter
- nur in 1 Datei sichtbar
- alle Funktionen static definieren, ausser jene, die die Schnittstelle nach aussen bilden

Speicherklassen lokaler Variabeln S.354

auto: auf Stack, wird automatisch hinzugefügt wenn sonst nichts steht
→ eigentlich sinnlos!

register: wird empfohlen (dem Compile) die Variable im Register anzulegen (schnellste Speicherart)
→ keine Garantie
→ compiler ist gut → braucht's meistens nicht!

static: nicht auf Stack, lokale static Variabeln sind nur im Block sichtbar, wo sie definiert wurden.
→ dies ist zu begrüssen → Code zu schützen vor Spionage!

Initialisierung: S. 361!

automatische Variabeln nicht vom System initialisiert, von definition sind sie undefiniert

Externe & statische lokale Variabeln automatisch auf 0, trotzdem explizit initialisieren
→ guter Programmierstil

Datei (File)

eine Datei ist eine logisch zusammenstellung von Daten, die zusammen behandelt werden und unter einen dem Betriebssysten bekannten Namen abgespeichert werden. Diese sind meist auf der Harddisk. Auch Geräte oder Schnittstellen können als Datei betrachtet werden.

Datenstrom ist ein Datensatz aus beliebig vielen Bytes. siehe Bild S. 370
Die Interpretation erfolgt über Anwendungsprogramme

Standard ausgänge & Eingänge

stdin	meist die Tastatur	Standardeingabe
stdout	meist die Konsole	Standardausgabe
stderr	meist die Konsole	Standard fehlerausgabe

Diese 3 Kanäle können in die Konsole umgelenkt werden

z.B. myprog > test.out

Umlenkung:

Standardausgabe	a > b	mache a und schreibe es in b
Standard eingabe	a < b	nehme aus a und mache b
Kombination	a < b > c	nehme aus a mache b und schreibe es in c

High/Low-Level Funktionen S. 387

High: standardisiert, portabel, printf(), scanf(), fprint(), fscanf(), fopen(), gets()
→ diese verwenden

Low: nicht standardisiert, nicht portabel, open(), creat(), write(), read(), lseek()
→ sind sehr schnell trotz dem nicht nutzen
nur für spezielle Betriebssysteme

High-Level-Dateizugriffsfunktion

nicht speziell zusammengefasst!
siehe im Buch S. 401 ff

→ alle Angaben ohne Gewähr!