D Einführung in die Programmiersprache C

D.1 C vs. Java

- Java: objektorientierte Sprache
  - ➤ zentrale Frage: aus welchen Dingen besteht das Problem
  - ➤ Gliederung der Problemlösung in Klassen und Objekte
  - ➤ Hierarchiebildung: Vererbung auf Klassen, Teil-Ganze-Beziehungen
  - ➤ Ablauf: Interaktion zwischen Objekten
- C: imperative / prozedurale Sprache
  - ➤ zentrale Frage: welche Aktivitäten sind zur Lösung des Problems auszuführen
  - ➤ Gliederung der Problemlösung in Funktionen
  - ➤ Hierarchiebildung: Untergliederung einer Funktion in Teilfunktionen
  - ➤ Ablauf: Ausführung von Funktionen

D.1 C vs. Java

1 C hat nicht

- Klassen und Vererbung
- Objekte
- umfangreiche Klassenbibliotheken

2 C hat

- Zeiger und Zeigerarithmetik
- Präprozessor
- Funktionsbibliotheken

D.2 Sprachüberblick

1 Erstes Beispiel (C-Programm unter Linux)

- Die Datei **`hello.c`** enthält die folgenden Zeilen:

```
/* say "hello, world" */
int main()
{
    printf("hello, world\n"); return 0;
}
```

- Die Datei wird mit dem Kommando **`cc`** übersetzt:

```
% cc hello.c            (C-Compiler)
oder
% gcc hello.c           (GNU-C-Compiler)
```

dadurch entsteht eine Datei **`a.out`**, die das ausführbare Programm enthält.

- ➤ ausführbares Programm liegt in Form von Maschinencode des Zielprozessors vor (kein Byte- oder Zwischencode)!

1 Erstes Beispiel (2)

- Mit der Option **`-o`** kann der Name der Ausgabedatei auch geändert werden – z. B.

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

- Das Programm wird durch Aufruf der Ausgabedatei ausgeführt:

```
% ./hello
hello, world
%
```

- Kommandos werden so in einem Fenster mit UNIX/Linux-Kommandointerpreter (Shell) eingegeben
  - ➤ es gibt auch integrierte Entwicklungsumgebungen (z. B. Eclipse)

2 Erstes Beispiel (C-Programm für AVR-Mikrocontroller)

- Die Datei **`red.c`** enthält die folgenden Zeilen:

```
/* switch red led on */
#include <led.h>
void main()
{
    sb_led_on(RED0); while(1);
}
```

- Die Datei wird mit dem Kommando **`avr-gcc`** übersetzt:

```
% avr-gcc -o red.elf -ffreestanding -mmcu=atmega32 ... red.c
```

➤ im Gegensatz zur Übersetzung eines Programms für Linux muss hier
– die Zielplattform angegeben werden *(Cross-Compilation)*
– Angaben über Bibliotheken und include-Dateien angegeben werden (...)

- Vereinfachung über "Makefile"

```
% make -f /proj/i4spic/pub/debug.mk red.elf
```


2 Erstes Beispiel (C-Programm für AVR-Mikrocontroller) (2)

- In der Datei **`red.elf`** liegt der ausführbare Programmcode für den Mikrocontroller vor
- dieser muss anschliessend auf den Mikrocontroller geladen werden
  - ➤ Mikrocontroller über USB-Schnittstelle an Entwicklungs-PC anschließen
  - ➤ Programm zum Übertragen des Codes (*Flashen*) starten

```
% make -f /proj/i4spic/pub/debug.mk red.elf.flash
```

- Weitere Details in den Übungen!

3 Aufbau eines C-Programms

- frei formulierbar - ***Zwischenräume*** *(Leerstellen, Tabulatoren, Newline und Kommentare*) werden i. a. ignoriert - sind aber zur eindeutigen Trennung direkt benachbarter Worte erforderlich
- ***Kommentar*** wird durch **`/*`** und **`*/`** geklammert
  keine Schachtelung möglich
- ***Identifier*** (Variablennamen, Marken, Funktionsnamen, ...) sind aus Buchstaben, gefolgt von Ziffern oder Buchstaben aufgebaut
  - "_" gilt hierbei auch als Buchstabe
  - Schlüsselwörter wie **`if`**, **`else`**, **`while`**, usw. können nicht als *Identifier* verwendet werden
  - ***Identifier*** müssen vor ihrer ersten Verwendung ***deklariert*** werden
- Anweisungen werden generell durch **`;`** abgeschlossen

4 Allgemeine Form eines C-Programms:

```
/* globale Variablen */
    ...

/* Hauptprogramm */
main(...)
{
    /* lokale Variablen */
        ...
    /* Anweisungen */
        ...
}

/* Unterprogramm 1 */
function1(...)
{
    /* lokale Variablen */
        ...
    /* Anweisungen */
        ...
}

/* Unterprogramm n */
functionN(...)
{
    /* lokale Variablen */
        ...
    /* Anweisungen */
        ...
}
```

5 wie ein C-Programm nicht aussehen sollte:

```
#define o define
#o ___o write
#o ooo (unsigned)
#o o_o_ 1
#o _o_ char
#o _oo goto
#o _oo_ read
#o o_o for
#o o_ main
#o o__ if
#o oo_ 0
#o _o(_,__,___)(void)___o(_,__,ooo(___))
#o __o(o_o_<<((o_o_<<(o_o_<<o_o_))+(o_o_<<o_o_)))
+(o_o_<<(o_o_<<(o_o_<<o_o_)))
o_(){_o_ _=oo_,__,___,____[__o];_oo ______;_____:____=__o-o_
_______:
_o(o_o_,____,__=(_-o_o_<___?_-
o_o_:___));o_o(;__;_o(o_o_,"\b",o_o_),__--);
_o(o_o_," ",o_o_);o__(--___)_oo
_______;_o(o_o_,"\n",o_o_);______:o__(_=_oo_(
oo_,____,__o))_oo _____;}
```

sieht eher wie Morse-Code aus, ist aber ein **gültiges** C-Programm.

D.3 Datentypen

- Datentypen →
  - ➤ Konstanten
  - ➤ Variablen

◆ Ganze Zahlen
◆ Fließkommazahlen
◆ Zeichen
◆ Zeichenketten

1 Was ist ein Datentyp?

- Menge von Werten
  +
  Menge von Operationen auf den Werten
  - ◆ **Konstanten** — Darstellung für einen konkreten Wert (`2`, `3.14`, `'a'`)
  - ◆ **Variablen** — Namen für Speicherplätze, die einen Wert aufnehmen können
  - ➥ Konstanten und Variablen besitzen einen Typ
- Datentypen legen fest:
  - ◆ Repräsentation der Werte im Rechner
  - ◆ Größe des Speicherplatzes für Variablen
  - ◆ erlaubte Operationen
- Festlegung des Datentyps
  - ◆ implizit durch Verwendung und Schreibweise (Zahlen, Zeichen)
  - ◆ explizit durch **Deklaration** (Variablen)

2 Standardtypen in C

- Eine Reihe häufig benötigter Datentypen ist in C vordefiniert

char	Zeichen (im ASCII-Code dargestellt, 8 Bit)
int	ganze Zahl (16 oder 32 Bit)
float	Gleitkommazahl (32 Bit) etwa auf 6 Stellen genau
double	doppelt genaue Gleitkommazahl (64 Bit) etwa auf 12 Stellen genau
void	ohne Wert

2 Standardtypen in C (2)

- Die Bedeutung der Basistypen kann durch vorangestellte ***Typ-Modifier*** verändert werden

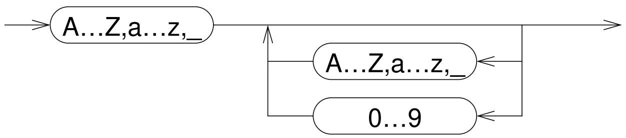
`short`, **`long`**
: legt für den Datentyp **`int`** die Darstellungsbreite (i. a. 16 oder 32 Bit) fest.
Das Schlüsselwort **`int`** kann auch weggelassen werden

`long double`
: **`double`**-Wert mit erweiterter Genauigkeit (je nach Implementierung) – mindestens so genau wie **`double`**

`signed`, **`unsigned`**
: legt für die Datentypen **`char`**, **`short`**, **`long`** und **`int`** fest, ob das erste Bit als Vorzeichenbit interpretiert wird oder nicht

3 Variablen

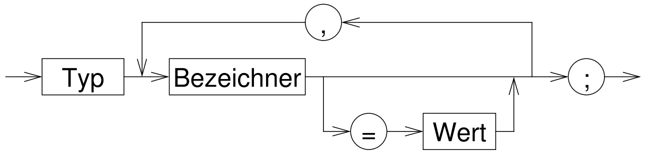
- Variablen haben:
  - **Namen** (Bezeichner)
  - Typ
  - zugeordneten Speicherbereich für einen Wert des Typs
    Inhalt des Speichers (= **aktueller Wert** der Variablen) ist veränderbar!
  - **Lebensdauer**
    wann wird der Speicherplatz angelegt und wann freigegeben
- Bezeichner



(Buchstabe oder _ ,
evtl. gefolgt von beliebig vielen Buchstaben, Ziffern oder _)

3 Variablen (2)

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



3 Variablen (3)

- Variablen-Definition: Beispiele

```
int a1;
float a, b, c, dis;
int anzahl_zeilen=5;
char Trennzeichen;
```

  - Position im Programm:
    - nach jeder "**{**"
    - außerhalb von Funktionen
    - neuere C-Standards und der GNU-C-Compiler erlauben Definitionen an beliebiger Stelle im Programmcode: Variable ab der Stelle gültig
- Wert kann bei der Definition initialisiert werden
- Wert ist durch Wertzuweisung und spezielle Operatoren veränderbar
- Lebensdauer ergibt sich aus der Programmstruktur