

Variablen

Variablen haben Namen, Typ und Geltungsbereich.

Name:

- für den Programmierer hilfreich; für die Semantik ohne Bedeutung (Umbenennung wäre möglich; z.B. var0, var1, var2, ...)

Typ:

- definiert mögliche Inhalte und Größe (z.B. 8-Bit-Zeichen, 32-Bit-Int-Zahlen, ...)
- definiert mögliche Operationen (Vergleiche, '+', '-', ...)

Geltungsbereich:

- für den Programmierer hilfreich; für die Semantik ohne Bedeutung (alle Variablen –bis auf Rekursion (später)– könnten auch global sein)

Variablen

1. Vereinfachung: Namen durch Nummer ersetzen:

Beispiel:

```
int global;
static int module;
void func(int parameter) {
    int func_auto;
    static int func_static;
    {
        int block_local;
        for (int for_local = 0; ...)
            ...
    }
}

int var0;
static int var1;
void func(int var2) {
    int var3;
    static var4;
    {
        int var5;
        for (int var6 = 0; ...)
            ...
    }
}
```

Variablen

2. Vereinfachung: Typ durch „generischen“ Typ (Sequenz von Bits) ersetzen

(Beispiel soll nur die Idee vermitteln; funktioniert in Hochsprache so nicht!)

```
int func(float x) {
    int y;

    y = 10 * sin(x);
    return y;
}

int func(bit32 x) {
    bit32 y;

    int(y) = 10 * sin(float(x));
    return y;
}
```

Variablen

3. Vereinfachung: Geltungsbereiche auflösen

```
int global;
static int module;

void func(int parameter) {
    int func_auto;
    static int func_static;
    {
        int block_local;
        for (int for_local = 0; ...)
            ...
    }
}

int global;
int module;
int parameter;
int func_auto;
int func_static;
int block_local;
int for_local;
void func() {
    for (for_local = 0; ...)
        ...
}
```

Variablen

4. Vereinfachung:

Man verwendet immer Variablen fester Bit-Länge (vielfach 8-Bit-Elemente „Bytes“).

- Variablen mit weniger Bits (z.B. vom Typ `boolean`) verschenken Speicherplatz.
- Variablen mit vielen Bits (z.B. 32-Bit-Integer-Variablen werden auf mehrere Teil-Variablen aufgeteilt. Variablen bekommen aufeinander folgende Nummern.

ggf. Aufteilen beim Schreiben, Zusammenfügen beim Lesen

Beispiel: 32-Bit-Integer-Variable wird aufgeteilt in 4 Byte-Variablen

```
10001011011100100110011001100111
<=> 10001011 - 01110010 - 01100110 - 01100111
```

Variablen

4. Vereinfachung: „große“ Variablen durch Sequenz von „kleinen“ ersetzen

(Beispiel soll nur die Idee vermitteln; funktioniert in Hochsprache so nicht!)

```
int func(short int x) {
    int y;

    y = 10 * x;
    return y;
}

int func(bit8 x0, bit8 x1) {
    bit8 y0, y1, y2, y3;

    {y0,y1,y2,y3} = 10 * {x0,x1};
    return {y0,y1,y2,y3};
}
```

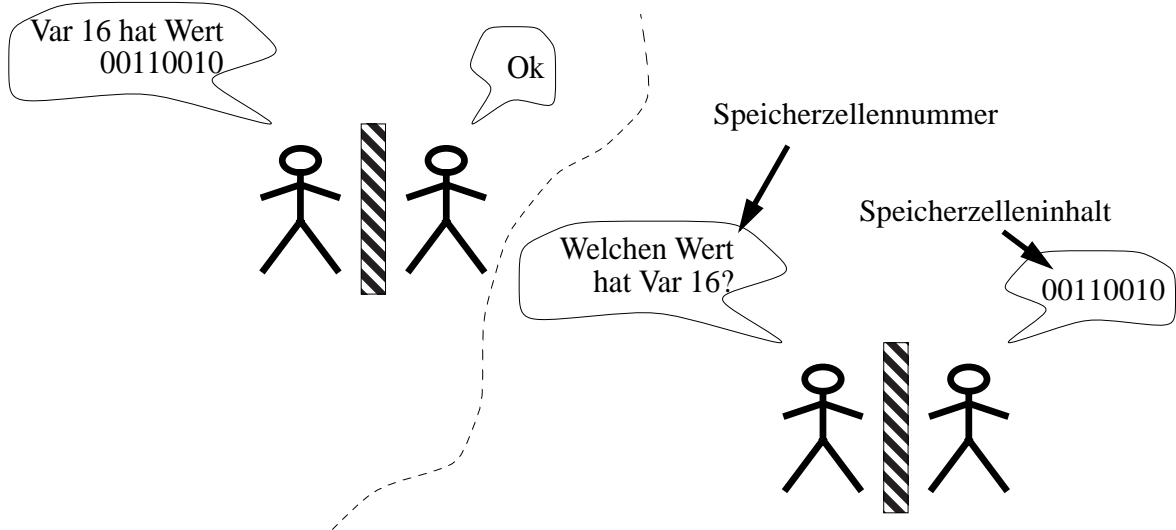
Variablen

Zum Speichern und Wiedergeben

von Variablen

reicht eine eindeutige Zahl als „Bezeichner“!

Variablen



Speicher

Speicher soll beliebige Daten aufnehmen können
(z.B. Zeichen, Integer-Zahlen, Floating-Point-Zahlen, Pointer, Programm-Code)

=> Aufnahme „anonymer“ Daten (Sequenz von '0' und '1')

Interpretation der Bit-Werte ist Programm-abhängig

Speicher (typische Werte)

Speicher organisiert in Einheiten (Maschinenwort) zu je N-Bits

- typisch 8-, 16-, 32- oder 64-Bit

typische Benennung:

- 8 Bits = 1 Byte
- 2 Bytes = 1 Word
- 2 Words = 1 Long (Double Word)
- 2 Longs = 1 Long long (Quad Word)

Aktuelle Speichergrößen (Beispiele):

- Mainframe: 64 GByte
- Workstation: 256 MByte
- Embedded Systems: 64 Byte

Speicher (Adressen/Bitnummern)

Einheiten besitzen Adressen (Hauptspeicher) bzw. Nummern/Namen (Register)

		7	6	5	4	3	2	1	0	Bitnummer
<i>Beispiel:</i>	0									
	1									
	2									
	3									
<i>Adresse</i>										
	1021									
	1022									
	1023									

Speicher (Adressen/Bitnummern)

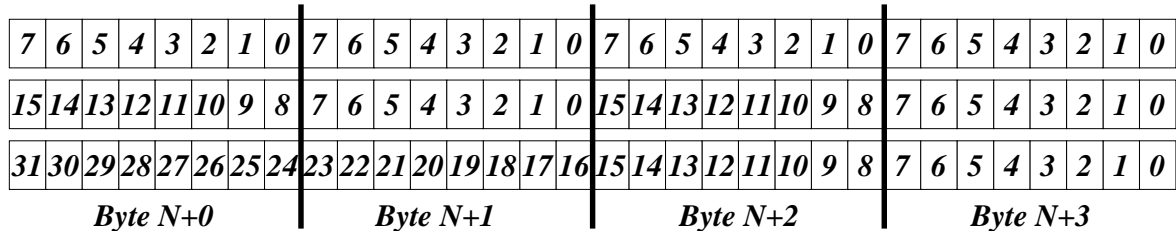
Beispiel für (Byte-) Adressen und Bitnummern eines 16-Bit-breiten Speichers:

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Bitnummer
	0																	
	2																	
	4																	
	6																	
<i>Adresse</i>																		
	1018																	
	1020																	
	1022																	

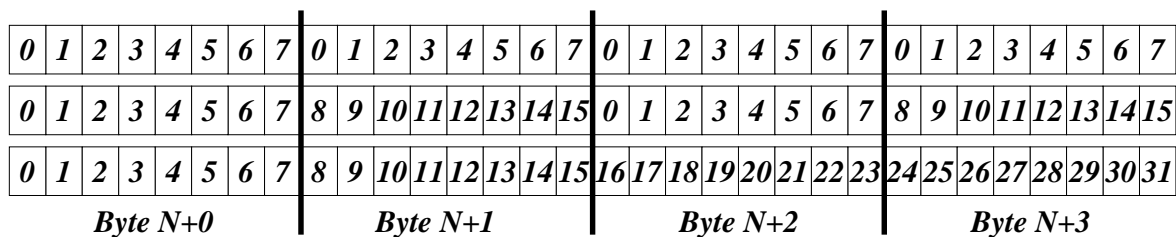
Speicher (Byte-Order)

Frage: Welche Bits werden im ersten, zweiten, usw. Byte gespeichert (Byte-Order)?

Big Endian (Most-Significant Byte First, Network-Byte-Order):



Little Endian (Least-Significant Byte First):



Speicher

Beziehungen zwischen

- Adress-Länge (A Bits),
- Speichergröße (S):

(Annahme: logisch adressiert werden jeweils Bytes)

Speichergröße:

$$S = 2^A \text{ Byte}$$

Konvention: 1 KByte = 1024 Byte, 1 MByte = 1024 KByte, ...

Speicher (einfache Datentypen)

Größen verschiedener einfacher Datentypen (typische Werte für C, C++, Java, Pascal):

- Character:	1 Byte	(z.T. 2 bzw. 4 Byte)
- Short Integer:	2 Byte	
- Integer	4 Byte	(z.T. 2 Byte)
- Long Integer:	4 Byte	
- Long Long Integer:	8 Byte	
- Single Precision Float:	4 Byte	
- Double Precision Float:	8 Byte	(z.T. 10 Byte)
- Pointer:	4 Byte	(z.T. 2 bzw. 8 Byte)

Speicher (symbolische Adressen)

Symbolische Adressen:

Original

äquivalente Schreibweise mit
symbolischen Adressen

	len	EQU 15
	from	EQU 16
	to	EQU 32
LDA #16		LDA #16
STA 15		STA len
LDX 15		LDX len
DECX		DECX
LDA 16,X		LDA from,X
STA 32,X		STA to,X
DECX		DECX
BPL ...		BPL ...

Speicher (symbolische Adressen)

Schreibweise (Beispiel m68x05):

```
x      EQU 192   ; "char"-Variable x hat Adresse 192
y      EQU 193   ; "short"-Variable y hat Adresse 193
z      EQU 195   ; "long"-Variable z hat Adresse 195
p      EQU 199   ; "pointer"-Variable p hat Adresse 199
array  EQU 201   ; Array hat Adresse 201
```

nachträgliches Einfügen oder Löschen von Variablen mühsam und fehleranfällig

äquivalente Schreibweise mit Pseudo-Operationen:

```
                ORG 192   ; Startadresse der Variablen
x      RMB 1
y      RMB 2
z      RMB 4
p      RMB 2
array  RMB 20
```

Speicher (symbolische Adressen)

Schreibweise i80x86-GNU-Assembler:

```
.data          /* Datensegment */
x:             .globl x   /* globale "char" Variable "x" */
              .zero 1
y:             .zero 2   /* lokale "short" Variable "y" */
z:             .globl z   /* globale "long" Variable "z" */
              .zero 4
p:             .zero 4   /* lokale Pointer-Variable "p" */
array:        .zero 20   /* 20-Byte großes Array "array" */
```

Read-Only Variablen auch im Programmsegment möglich.

Speicher (strukturierte Datentypen)

Moderne Programmiersprachen kennen

- Strukturen / Records
- Arrays

sowie Kombinationen davon.

Zur Speicherung von Variablen dieser Typen müssen mehrere (normalerweise aufeinander folgende) Speicherzellen verwendet werden.

Beispiel:

- zur Speicherung eines Characters benötigt man ein Byte
- zur Speicherung eines Arrays bestehend aus N Characters benötigt man N Bytes (die Struktur geht verloren)

Speicher (mehrdimensionale Arrays)

Speicher ist „eindimensionales“ Array

Problem: wie speichert man mehrdimensionale Arrays?

*Array mit N Zeilen mit jeweils M Elementen enthält insgesamt N*M Elemente*

*diese werden von 0 bis N*M-1 durchnummeriert => eindimensionales Array*

zweidimensionales Array

```
int f1[10][5];  
int i;  
int j;  
  
... = f1[i][j];
```

eindimensionales Array

```
int f2[10*5];  
int i;  
int j;  
  
... = f2[i*5 + j];
```

Speicher (Beispiel)

C-Struktur

```
struct {
    char c;          /* 1 Byte */
    long int i;     /* 4 Byte */
    double f;       /* 8 Byte */
} x;
```

Abbildung auf Speicher

Adresse	Inhalt
N + 0:	x.c
N + 1:	x.i (erstes Byte)
N + 2:	x.i (zweites Byte)
N + 3:	x.i (drittes Byte)
N + 4:	x.i (viertes Byte)
N + 5:	x.f (erstes Byte)
...	
N + 12:	x.f (achtes Byte)

Speicher (Beispiel)

C-Struktur:

```
struct {
    char c;          /* 1 Byte */
    long int i;     /* 4 Byte */
    short int f;    /* 2 Byte */
} x;
```

GNU-Assembler für i80x86:

```
        .data
x:      .global x
        .zero 7
```

Speicher (Alignment)

Zugriff auf 2-Byte-Werte (bzw. 4-Byte-, 8-Byte-Werte) schneller, wenn diese an geraden (bzw. durch 4, 8 teilbaren) Adressen beginnen.

(manche Architekturen erfordern korrektes Alignment – sonst „Bus Error“)

=> Alignment

Erreichbar durch das (automatische) Einfügen von „Dummy“-Variablen.

Beispiel:

Ohne Alignment:

```
/* 4*N+0 */ char c;  
/* 4*N+1 */ short s; /* kein passendes Alignment */
```

Mit Alignment:

```
/* 4*N+0 */ char c;  
/* 4*N+1 */ char dummy;  
/* 4*N+2 */ short s; /* passendes Alignment */
```

Speicher (Operationen)

Folgende Speicher-Operationen stehen i.A. zur Verfügung:

```
datum = load(address);
```

```
store(address, datum);
```

Die load- bzw. store-Funktionalität des Speichers ist auch per CPU-Operation verfügbar

- direkt (um Daten aus dem Hauptspeicher in die CPU zu holen bzw. von der CPU in den Speicher zurückzuschreiben)
- als Teil von komplexeren CPU-Operationen, die Daten aus dem Hauptspeicher lesen bzw. Daten im Hauptspeicher ablegen wollen

Register

Ähnlich Speicherzellen; aber:

- nur wenige Register verfügbar
- schnellerer Zugriff

=> meist für temporäre Variablen (einfache Datentypen) verwendet

Register

Register der i80x86 CPU:

- %eax, %ebx, %ecx, %edx, %edi, %esi, %ebp (je 32 Bit)
z.T. auch als %ax, %bx, ... ansprechbar - 16 Bit
oder auch als %ah/%al, %bh/%bl, ... - 8 Bit
- Stack-Pointer (%esp) (32 Bit)
- Condition-Codes/Flags-Register
- Program-Counter (32 Bit)
- ...

Register des m68x05 Mikro-Controllers:

- Akkumulator (A) (8 Bit)
- Index-Register (X) (8 Bit)
- Stack-Pointer (5 Bit)
- Condition-Codes/Flags-Register (5 Bit)
- Program-Counter (12 Bit)

Speicher- / Register-Zugriffe (i80x86)

Zugriff auf Variablen im Speicher

- 1-Byte-Werte: `movb address, ...; movb ..., address`
- 2-Byte-Werte: `movw address, ...; movw ..., address`
- 4-Byte-Werte: `movl address, ...; movl ..., address`

Zugriff auf Variablen in Registern

- 1-Byte-Werte: `movb %reg, ...; movb ..., %reg`
- 2-Byte-Werte: `movw %reg, ...; movw ..., %reg`
- 4-Byte-Werte: `movl %reg, ...; movl ..., %reg`

Beispiel (i80x86):

```
movw %ax, 16 # Kopiere short-Wert im Akku nach Adresse 16
movl 4, %esp # Kopiere long-Wert von Adresse 4 in SP
```

Speicher- / Register-Zugriffe (m68x05)

Transfer Speicher / Register

- 1-Byte-Werte: `lda address; stx address`
- 2-Byte-Werte: -
- 4-Byte-Werte: -

Beispiel (m68x05):

```
sta 16      ; Kopiere Byte-Wert im Akku nach Adresse 16
lda 4       ; Kopiere Byte-Wert von Adresse 4 in Akku

ldx 7       ; Kopiere Byte-Wert von Adresse 7 ins Index-Reg.
stx 9       ; Kopiere Byte-Wert im Index-Reg. nach Adresse 9
```