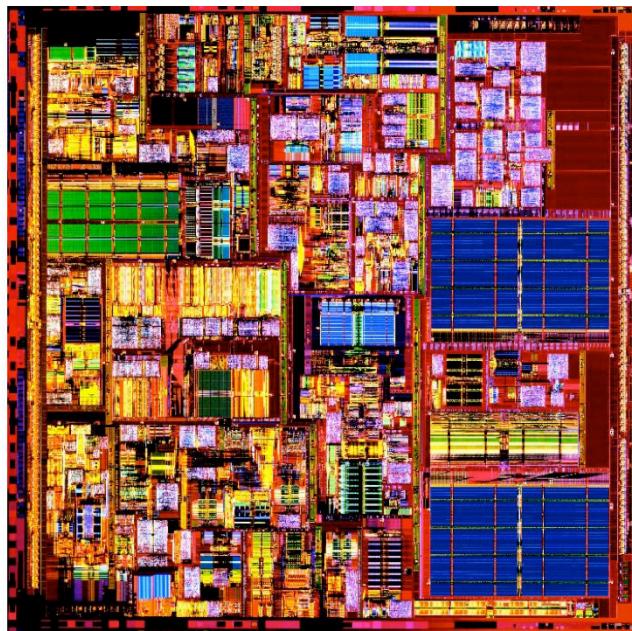


Rechnerarchitektur



Rechnerarchitektur

1

Adresse

Prof. Dr. Paolo Favaro
Institut für Informatik
Universität Bern
Neubrückstrasse 10, Raum 101
3012 Bern

Telefon: 031 631 44 51
Email: pao.lo.favaro@inf.unibe.ch
WWW: <http://cvg.unibe.ch>

Termine

- Vorlesung: dienstags von 13.15-15.15 Uhr
- Übungen: dienstags von 15.15-16.15 Uhr
- Sprechstunde: nach Vereinbarung (email schicken)

Inhalt

1. C Einführung
2. Sprache des Rechners
3. Performance
4. Prozessorarchitektur
5. Pipelining
6. Speicherhierarchie
7. Ein- und Ausgabe

Literatur

- ❑ D. Patterson, J. Hennessy: Rechnerorganisation und -entwurf, Elsevier
- ❑ Die relevanten Kapitel sind im **Ilias** online verfügbar.
- ❑ Auf diverse C Bücher kann ebenfalls zugegriffen werden.

Einführung in C



Literatur und Referenz

- Kernighan & Ritchie: Programmieren in C, zweite Ausgabe ANSI C
- C man pages sind auf Unix Systemen Standard !
 - Mit „man <Funktionsname>“ erhält man Informationen zu allen Standard C Funktionen.
 - Achtung! Häufig gibt es die gleiche Funktion auch in anderen Programmiersprachen. Also darauf achten, dass man wirklich die C man page angezeigt bekommt (man -a)!

Historisches

- C entwickelt als Programmiersprache für das Unix Betriebssystem
- „Programmieren in C“ erstmals 1978 in USA erschienen und beschreibt K&R C Standard.
- Seit 1983 neuer Standard „ANSI C“
- Später objektorientierte „Erweiterung“ C++ von Bjarne Stroustrup
- Die Mehrheit aller modernen Betriebssysteme sind in C/C++ geschrieben.
- C Syntax war die Vorlage für viele Programmiersprachen, z.B. für Java.

Das erste Programm

- **#** sind Präprozessor Anweisungen
- **#include <>** veranlasst den Präprozessor, das angegebene file einzufügen.
- **#include <stdio.h>** fügt z.B. Informationen über I/O Bibliothek ein.
- **main(...)** ist die Funktion die bei Programmstart aufgerufen wird.
- **printf(...)** dient zur Ausgabe von einfachen Strings bis hin zu (sehr) komplexen Ausdrücken.

```
#include <stdio.h>
main()
{
    printf("hello world\n");
}
```

Ausgabe:

```
hello world
```

Are preprocessor instructions
include <> causes the preprocessor to include the specified file.
include includes information about the I / O library.
main (...) function is called when the program starts.
printf (...) is used to output simple strings to (very) complex expressions.

Variablen und Arithmetik

- Kommentare werden durch `/* */` geklammert
- Vereinbarung von Variablen am **Anfang** eines Blocks.
- Variablen werden **nicht** automatisch initialisiert !
- Alternativ zu `fahr=fahr+step`
`fahr+=step;`

```
#include <stdio.h>
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;           /* untere Grenze */
    upper = 300;          /* obere Grenze */
    step = 20;            /* Schrittbreite */

    fahr=lower;
    while(fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf(" %d    %d\n", fahr, celsius);
        fahr=fahr+step;
    }
}

Ausgabe:
0          -17
20         -6
...
300        148
```

Rechnerarchitektur

10

Declaration of variables at the beginning of the blocks

There is a better way than specifying all the limits as variables inside the code (see next code)

#define

- Keine Variablen, eher eine Art „search-replace Mechanismus“
- Findet im Präprozessor **vor** dem eigentlichen Compilieren statt.
- Sehr praktisch für Konstanten.

```
#include <stdio.h>

#define L_LIMIT 0
#define U_LIMIT 300

main()
{
    int fahr, celsius, lower, upper, step;

    lower = L_LIMIT; /* untere Grenze */
    upper = U_LIMIT; /* obere Grenze */
    step = 20; /* Schrittbreite */

    fahr=lower;
    while(fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf(" %d      %d\n", fahr, celsius);
        fahr=fahr+step;
    }
}

Ausgabe:
 0      -17
 20     -6
 ...
 300    148
```

Rechnerarchitektur

11

Not a variable but rather a type of search and replace mechanism

The preprocessor looks for these expressions and converts all of them before the compiler st

Programmstruktur

```
#include <stdio.h>

#define L_LIMIT 0
#define U_LIMIT 300

int pepe;           /* globale Variable */

main()
{
    int fahr, celsius, lower, upper, step;

    lower = L_LIMIT; /* untere Grenze */
    upper = U_LIMIT; /* obere Grenze */
    step = 20;        /* Schrittbreite */

    fahr=lower;
    while(fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf(" %d    %d\n", fahr, celsius);
        fahr=fahr+step;
    }
}

Ausgabe:
0          -17
20         -6
...
```

Includes & Defines

globale Variablen
globale Deklarationen
(structs, typedefs)

Funktionsdeklarationen

Funktionen

Rechnerarchitektur

12

Fkt. Deklarationen kommen später...

Elementare Datentypen

- char ein einzelnes Zeichen oder eine 8 bit Zahl, je nachdem
- float Fliesskommazahl
- double Fliesskommazahl
- int ganzzahliger Wert („natürliche Grösse“)
- short ganzzahliger Wert (mindestens 16 bit)
- long ganzzahliger Wert (mindestens 32 bit)

- „signed & unsigned“ Modifizierer bei Integer/char Datentypen legen Wertebereich fest.
- Abgesehen von char sind die Grössen aller Datentypen plattformabhängig

An individual symbol or an 8 bit number, as the case may be.

Natural size refers to the size that best fits the processor where the data is used.

"Signed & unsigned" modifier for integer / char data types determine value range.

Apart from the sizes of all char data types are platform-dependent

Arrays

- Beispiel:
 - char name[20]; int zahlen[100]; double pepe[2];
- char Arrays werden in C zum Speichern von Strings verwendet, können aber auch Zahlenreihen sein.
- Ein char Array x , der den String „OK“ enthält:
char X[3];
X[0]='O'; X[1]='K'; X[2]=0;
- Mehrdimensionale Arrays x[3][3] möglich.
- Es findet keine Überprüfung von Arraygrenzen statt !

```
int i,x[10];
for(i=0;i<100;++i)x[i]=42; /* ist legal */
```

example:

char name [20]; pay int [100]; pepe double [2];

char arrays are used in C for storing strings, but can also be number sequences.

A char array x contains the string "OK":? char X [3];? X [0] = 'O', X [1] = 'K', X [2] = 0;

Multidimensional arrays x [3] [3] possible.

Check of array bounds does not take place/happen (automatically).

printf(...)

- Eine der mächtigsten Funktionen in C.
- `printf(formatstring, args, ...);`
- *formatstring* entweder nur „hello world“ oder Lückentext mit Platzhaltern (z.B. %d). Platzhalter werden durch *args* ersetzt.
- Anzahl und Typ der Platzhalter in *formatstring* muss mit Anzahl der *args* übereinstimmen!
- Platzhalter sind z.B: %d, %f, %u, %x, %X
- `printf("X %d, %x, %X, %f %c %c\n", 255, 255, 255, 2.0, 'a', '98);`
---> Ausgabe: X 255, ff, FF, 2.0, a, b
- Analog dazu scanf für formulierte Eingabe !

%d, %i dezimal mit Vorzeichen (sign)

%u dezimal ohne Vorzeichen

Numbers and placeholders must match in number!

Similarly, scanf for formatted input!

One of the most powerful features in C.

`printf (format string, args ...);`

format string either "hello world" or fill in the blank with placeholders (eg% d). Placeholder:

Number and type of placeholders in string format must match the number of args match!

Wildcards are e.g.:% d,% f,% u,% x,% X

`printf ("X% d,% x,% X,% f% c% c \ n", 255,255,255,2.0, 'a', 98);?` ---> Output: X 255, ff, FI

Zuweisungen



`y = x = 10;`

`y hat den Wert 10
x hat den Wert 10`

`x = 6 + (y = 4 + 5);`

`y hat den Wert 9
x hat den Wert 15`

Assignments

is evaluated as a value

evaluates to the same value

Prioritäten von Operatoren

- Priorität bestimmt die Reihenfolge.

- Beispiel:

- != ist stärker als =
 - c = getchar() != EOF entspricht
c = (getchar() != EOF)

```
#include <stdio.h>
main() {
    int ch;
    while( (ch = getchar()) != EOF )
        putchar(ch);
}
```

- Kann sehr sehr kompliziert werden,
also im Zweifelsfall (oder besser
immer) klammern!

Typumwandlung

- Implizit wird bei arith. Operationen immer in den „höheren“ Datentyp umgewandelt

```
int i; double d,e;  
i=1; e=2.0;  
d=i/100;           /* d ist 0 */  
d=i/100.0;         /* d ist 0.01 */  
d=i/e;             /* d ist 0.5 */
```

- Cast Operator erlaubt explizite Typumwandlung

```
int i; double d;  
i=1;  
d=(double)i/100;    /* d ist 0.01 */  
d=((double)i)/100;  /* d ist 0.01 */
```

Typumwandlung hat höhere Priorität als Division.

Die beiden Zeilen sind also gleich und d ist 0.01

Typecast

Implicit in the arithmetic

Higher level datatypes dictate rule

Cast operators allow explicit type casting

Operatoren (Übersicht)

- $*$, $/$, $\%$ Multiplikation, Division, Modulo
- $+$, $-$ Addition, Subtraktion
- $<<$, $>>$ bitshift links und rechts
- $<$, $>$, $<=$, $>=$ Vergleich
- $==$, $!=$ Gleichheit, Ungleichheit
- $\&$ bitweise AND
- $|$ bitweise OR
- \sim Bitkomplement
- $\&\&$ logisches AND
- $\|$ logisches OR
- $=, +=, -=, *=, /=, \% =, <<=, >>=, !$
 $=, \&=$ Zuweisung
- $++, --$ Inkrement, Dekrement
- $'c'$ liefert ASCII Wert des Zeichens c
- $?:$ bedingt „ $a=b?1:2$ “
- $\text{sizeof}(\text{Vartyp})$
Speicherbedarf einer Variable in Byte

Bei Bitshift wird mit Nullen aufgefüllt. D.h Shift entspricht multiplikation (division) mit 2.

? - Operator

```
if ( x == 1 )
    y = 10
else
    y = 20;
y = (x == 1) ? 10 : 20;
```

```
if (x==1)
    puts("take car")
else
    puts("take bike");
(x == 1) ? puts("take car") : puts("take bike");
oder
puts( (x == 1) ? "take car" : "take bike");
```

Inkrement und Dekrement Operatoren.

- `++` und `--` erhöhen/erniedrigen Variable um einen Wert.
 - Position vor oder hinter der betreffenden Variable entscheidet in zusammengesetzten Ausdrücken darüber, wann Operation ausgeführt wird.
 - Reihenfolge der Argumentbearbeitung bei Funktionsaufrufen ist kompilerabhängig !!
- ```
#include <stdio.h>
main()
{
 int i=0, j=0, x[10];
 printf("%d\n", i++); /* 0 */
 printf("%d\n", i); /* 1 */
 printf("%d\n", ++i); /* 2 */

 j=i++;
 printf("%d %d\n", i, j); /* 3 2 */

 j=++i;
 printf("%d %d\n", i, j); /* 4 4 */

 printf("%d %d\n", i++, i+1); /* !!!! */
 /* z.B. Arrayelemente auf 0 setzen */
 for(i=0;i<10;x[i++]=0);
}
```

`++` And `-` to increase / decrease variable by a value.

Position before or behind? Decides the relevant variable in compound expressions know what?

Order of argument processing function calls is compiler dependent!

## Kontrollstrukturen

```
int i=0;
• for(...;...;...){...;} while(i<10){
 printf("%d\n",i);
 ++i;
• do{...;}while(...); }

• while(...){...;} for(i=10;i<20;++i){
 printf("%d\n",i);
 }

• if(...){...;} else {...;} for(i=10;i<20;){
 printf("%d\n",i++);
 }

• break, continue for(i=10;i<20;printf("%d\n",i++));

 do{
 i=....; /* mache was mit i */
 }while(!i);

 while(1){
 ...; /* mache was mit i */
 if(++i>10)break;
 }
```

Rechnerarchitektur

22

Die **continue** Anweisung springt weiter zum Anfang der Schleife, überspringt also die Anwe

## **switch-case**

---

- Ersetzt mehrere if-else Bedingungen
- Kann nur für integer / char Variablen verwendet werden

```
switch(n){
 case 0: printf(„n ist 0\n“); break;
 case 1: printf(„n ist 1\n“); break;
 default: printf(“n ??\n“);
}
```

Replace multiple if-else conditions

Can only be used for chars and integers

## Strukturen

- Struct ist ein zusammengesetzter Datentyp

```
struct {int a; int b; char n[32];} s,w;
s.a=1; s.b=2; s.n[0]='b'; w.a=1; w.b=2;
```
- Meistens mit Structure-Tag verwendet.

```
struct Point {int x; int y;};
struct Point a,b;
a.x=0; b.y=1;
```
- Daten werden 1 zu 1 auf Speicher abgebildet !!
- Beispiel

```
#include <stdio.h>

struct Point {int x; int y;};

main(){
 struct Point a,b;
 a.x=1; a.y=2;
 printf("Punkt(%d/%d)\n",a.x,a.y);
}
```

Compound data type

Mostly used with struct tag

Data mapped 1-1 in memory!

## Unions

- Union analog zu struct, allerdings teilen sich die Elemente innerhalb der Union den selben Speicherbereich.

```
union {char x[4]; long b;} u;
```

- Eine Veränderung an **u.x[0]** hat also eine Veränderung von **u.b** zur Folge.

- Beispiel:**

```
struct conditions {
 float temp;
 union feels_like {
 float wind_chill;
 float heat_index;
 }
 today;
```

Union analog to struct, however, share elements within the same memory area

A change of **u.x[0]** has an effect in **u.b**

(overlapping variables)

## Bitfeld

- Analog zu struct, aber mit Angabe der Breite (in bits) der einzelnen Elemente
- Beispiel: IP Header (i386)

```
struct Packet{
 unsigned ihl:4;
 unsigned version:4;
 unsigned tos:8;
 unsigned len:16;
 unsigned id:16;
 unsigned frag:16;
 unsigned ttl:8;
 unsigned prot:8;
 unsigned crc:16;
 unsigned source:32;
 unsigned dest:32;
 char data;};
```

Analog to struct, but with an indication of the number of bits for each field (element)

## Typedef

- „Umbenennen“ eines Variabtentyps
- Meistens mit structs, unions und bitfeldern

```
. #include <stdio.h>
typedef struct {int x; int y;} Punkt;
typedef int Fritz;
main(){
 Punkt a;
 Fritz b;
 a.x=10; a.y=20;
 b=1;
}
```

Rename a variable type

Mostly with struct and unions and bit fields

## Funktionen

- Funktionen müssen vor ihrem ersten Aufruf bekannt sein und daher entweder definiert oder deklariert werden.
- Funktionen haben einen Rückgabewert (default ist int)
- Bei der Parameterübergabe werden die **Werte** der Parameter übergeben, nicht die Parameter selber!  
Ändert man den Wert innerhalb der Funktion, so ändert sich am Wert ausserhalb der Funktion nichts.

Must be known before being called

Therefore must be defined or declared

Have a return value

Passing parameters by value not parms themselves

Variables inside do not change once outside

## Funktionen

```
#include <stdio.h>

typedef struct {char *name; char *vorname;} Person;

void f(Person t){

 t.name="Meier";

}

main(){

 Person p;

 p.name = "Studer";

 p.vorname = "Thomas";

 f(p);

 printf("%s\n",p.name);
}
```

Ausgabe auf dem Bildschirm:  
Studer

## Funktionsdefinition und -deklaration

- Funktionsdefinition

```
#include <stdio.h>

int f(int z)
{
 return z*z;
}

int main()
{
 printf("%d\n",f(17));
}
```

- Funktionsdeklaration

```
#include <stdio.h>

int f(int);
int main()
{
 printf("%d\n",f(17));
}

int f(int z)
{
 return z*z;
}
```

## **Pointer**

---

- Pointer ist eine Variable, die die Adresse einer anderen Variablen enthält.
- Pointer haben einen Typ, abhängig davon wohin sie zeigen:
  - Zeiger auf char
  - Zeiger auf long
  - Zeiger auf struct
  - (auch Zeiger auf Funktionen möglich)
  - Zeiger auf Zeiger auf ...
  - Zeiger zeigen häufig auf einen Block von Variablen
- Operatoren:
  - \* dereferencing & addressoperator
  - nicht verwechseln mit Multiplikation und bitweisem & !!!

## Pointer

|         |      |      |      |      |      |      |
|---------|------|------|------|------|------|------|
| Adresse | 1000 | 1001 | 1002 | 1003 | 1004 | 1005 |
| Inhalt  | 1004 |      |      |      | 65   |      |

```
int *z_x;
int x;
x = 65;
z_x = &x;
```

Der Wert der Variablen x ist in der Speicherzelle 1004 abgelegt

Der Wert der Variablen z\_x ist in der Zelle 1000 gespeichert.

z\_x enthält die Adresse des Wertes von x.

Value of x stored in memory cell 1004

Z\_x contains the address

## Pointer

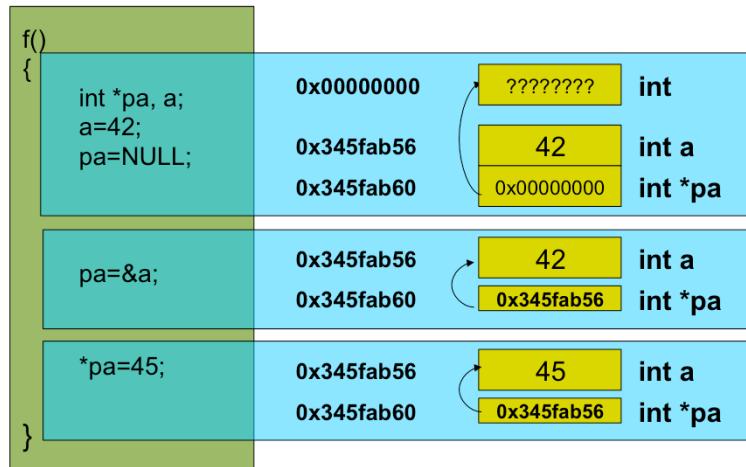
- Pointertypen werden durch \* gebildet
  - **char \*a**      *Pointervariable a, die auf char zeigen soll*
  - **int \*a,\*b**    *Pointervariablen a und b, beide auf int*

- Beispiel

```
main(){
 int a,b;
 int *pa, *pb;
 pa=&a; pb=&b; /* Adressoperator /
 a=1; b=2;
 printf("%d %d %d %d\n",a,b,*pa,*pb);
 /* 1 2 1 2 */
}
```

Pointers are formed by \*

## Pointer im Schema



## Pointer und Arrays

- Pointer zeigen häufig nicht nur auf eine einzelne Variable, sondern auf eine Reihe gleicher Variablen (ein Array). Letztendlich bedeutet int a[16] nur einen Zeiger int \*a, der auf einen reservierten Speicherblock aus 64 Bytes (16 \* sizeof(int)) zeigt.

```
main(){
 int a[16]; /* a ohne [] hat den Typ int* */
 ...
```

- Der Offsetoperator [ ] zählt den entsprechenden Wert zu dem in a gespeicherten Adresswert hinzu und greift dann auf die entsprechende Speicherstelle zu.



Pointers often point not just to a single variable, but rather to a list of similar variables (an array).

Ultimately it means that

Int a[16] is only a pointer int \*a for which we reserved a block in memory of 64 bytes (4 bytes).

The offset operator counts the appropriate value of the address and takes to the appropriate array element.

## Pointer Arithmetik

- Pointer enthalten Adresse d.h. eine Zahl. Die Zahl kann man verändern und somit den Pointer an eine andere Stelle zeigen lassen.

```
main(){
 char *s="Hello World";
 char *p;

 for(p=s; *p!=0; p++)printf("%c\n", *p);
 for(p=s; *p; p++)printf("%c\n", *p);
}
```

- Achtung: Rechenoperationen bei Pointern beziehen sich immer auf die Breite des VariablenTyps! Im oberen Fall ist diese Breite 1. Wäre p ein Zeiger auf int, so würde ++ den Adresswert um sizeof(int) Bytes erhöhen.

Pointers contain the address, i.e. a number.

The value can be changed and then it will point to another memory address

Warning: the arithmetical operations on pointers always relate to the type the pointer points to.

For example if p were a pointer to an integer then the ++ increments p of the size of an int (4 bytes).

0 heisst false,

ungleich 0 heisst true.

Also statt (a != 0) einfach a als Bedingung (condition) setzen.

## Pointer Arithmetik (Beispiele)

- ```
main(){
    double v[20];
    double *p;

    for(p=v;p-v<20;p++)
        printf("%f\n",*p);

    for(p=v;
        (unsigned long)p-(unsigned long)v<20*sizeof(double);
        p++)
        printf("%f\n",*p);
}
```
- **p++ erhöht Zahlenwert von p um sizeof(double) und nicht um 1!**
- Das Ergebnis der Subtraktion p-v ist also in sizeof(double) Einheiten.
- Gilt auch für alle anderen arithmetischen Operatoren

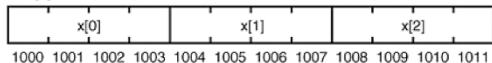
P++ increased the value by sizeof(double) not 1!

The result of the subtraction p-v is also in sizeof(double) units (Einheiten)

The same applies to other math operators

Pointer Arithmetik (Beispiele)

int x[3];



double ausgaben [2];



1: $x == 1000$

2: $\&x[0] == 1000$

3: $\&x[1] = 1004$

4: $ausgaben == 1240$

5: $\&ausgaben[0] == 1240$

6: $\&ausgaben[1] == 1248$

Pointer Arithmetik con't

```
main() {  
    int x, y;  
    x = 10;  
    printf ("x = %d\n", x);  
    *((&y)+1) = 20;  
    printf ("x = %d\n", x);  
}  
  
y und x sind auf hintereinanderfolgenden  
Speicherplätzen abgelegt!
```

Output:

x = 10
x = 20

Zuerst y , dann x

Y and x are consecutive in the memory!

Pointer Arithmetik con't

```
main() {  
    int x,a [10], i ;  
    x = 10;  
    printf ("x = %d\n",x);           //x=10  
    for ( i=0; i<=15; i++) a[i]=20;  
    printf ("x = %d\n", x);         //x=20  
}
```

Keine Tests auf Array-Grenzen

Zuerst i, dann a[10], dann x

No test on the array limits

Vergleiche Java

```
public class Array1 {  
  
    public static void main (String args []) {  
        int x, a [] = new int[10], i ;  
        x = 10;  
        System.out.println ("x=" + x);  
        for ( i=0; i<=15; i++) a[i]=20;  
        System.out.println ("x=" + x);  
    }  
}  
  
x=10  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 10  
at Array1.main(Array1.java:7)
```

Comparison to Java

Pointer und Strukturen

- Hat man, statt der Struktur selber, einen Pointer auf die Struktur, so muss man statt des . den Operator -> verwenden um auf Elemente der Struktur zuzugreifen.
- ```
int f(struct Point *p)
{
 printf("(%d/%d)",p->x,p->y);
 return 0;
}

int f(struct Point p)
{
 printf("(%d/%d)",p.x,p.y);
 return 0;
}
```
- Gilt natürlich auch für bitfelder und unions !

One has instead of the structure itself, a pointer to the structure, so one must instead of the

the operator

->

use to access the elements of the structure

The same goes for bitfields and unions!

Same as (\*p).x and (\*p).y but shorter notation

## Pointer und Funktionen

```
int f(int *x){
 *x=17;
 return 0;
}

main(){
 int i=0;
 f(&i);
 printf("%d\n",i); /* 17 */
}
```

**Pointer erlauben so Funktionen mit mehreren „Rückgabewerten“.**

Vergleiche Folie 24

Pointers allow functions to return multiple „return values“

## void\*

```
void haelfte(void *x, char typ) {
 /* Je nach Wert von typ wird der Zeiger x */
 /* entsprechend umgewandelt und durch 2 geteilt. */
 switch (typ) {
 case 'i': {
 *((int *)x) /= 2;
 break; }
 case 'l': {
 *((long *)x) /= 2;
 break; }
 case 'f': {
 *((float *)x) /= 2;
 break; }
 case 'd': {
 *((double *)x) /= 2;
 break; }
 }
}
```

Zeiger auf void erlauben die  
Übergabe von beliebigen  
Datentypen.  
Zeiger auf void können aber  
nicht dereferenziert werden.  
Ein Cast ist notwendig.

Depending on the value type of the pointer X  
Converted accordingly and divided by 2

A pointer to void allows the transfer to any other type  
but a cast is required

## **Dynamischer Speicher**

- Bislang nur statischer Speicher.
- Funktionen für dynamischen Speicher:
  - malloc( Grösse in Bytes );
  - calloc( Anzahl der Elemente, Grösse eines Elementes);
  - free(Zeiger auf Speicherblock);
- malloc und calloc liefern einen Zeiger auf einen Speicherblock zurück. Bei Fehler ist der Rückgabewert der Nullpointer „NULL“;
- free() gibt Speicherblock wieder frei.

Dynamic memory

So far, only static memory

Functions for dynamic memory

Malloc(size in bytes)

Calloc(number of elements, size of the largest element)

Free (pointer to memory block)

Malloc and calloc return a pointer to a block of memory

On error they return the NULL pointer

Free frees memory

## Dynamischer Speicher: Beispiel

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

main(){
 int *a, i;
 a=malloc(1024*sizeof(int)); /* 1024 ints */
 /* a=calloc(1024,sizeof(int)); */
 if(a==NULL){
 printf("no more memory");
 exit(1);
 }
 for(i=0;i<1024;i++)a[i]=0; /* array auf 0 setzen */
 /* do something with a */
 free(a);
}
```

- malloc gibt einen Zeiger auf void zurück. Damit können alle Datentypen im reservierten Speicher abgelegt werden.
- NULL ist eine definierte Konstante aus stdlib.h. Sie zeigt an, dass ein Zeiger nicht initialisiert ist.

Malloc gives a pointer to void

This allows all data types to be reserved in memory

NULL is a constant defined in stdlib.h

It indicates that a pointer is not initialized

## Dynamischer Speicher: Pitfall

```
main() {
 f();
 free(a); /* Fehler, es gibt hier kein a */
}

int f()
{
 char *a;
 a=malloc(256);

 /* do something with a*/
 return 0;
}
```

- Hat man einmal den Pointer auf den allozierten Speicherblock verloren, so kann man den Speicher nicht mehr freigegeben.
- **Dynamisch allozierter Speicher wird nicht automatisch freigegeben!**

Once you have lost the pointer to a block of memory, you cannot release it anymore

Allocated dynamic memory is not freed automatically!

## **Zeiger auf Funktionen**

```
float quadrat(float x); /* Der Funktionsprototyp. */
float (*p)(float x); /* Die Zeigerdeklaration. */
float quadrat(float x) /* Die Funktionsdefinition. */
{
 return x * x;
}
```

### **Aufruf mit Funktionenzeiger:**

```
p = quadrat;
antwort = p(x);
```

Mit Funktionenzeigern können Funktionen als Argumente an andere Funktionen übergeben werden. Beispielsweise kann eine Vergleichsfunktion Argument einer Sortierfunktion sein.

Pointers to functions

Function Prototype

Pointer declaration

Function definition

Call with function pointers

With function pointers one could pass another function as argument

For example a comparison function argument could be a sorting algorithm

## Zeiger auf Funktionen (Bsp)

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,
 int (*cmp)(const char *, const char *));
int main(void) {
 char s1[80], s2[80];
 int (*p)(const char *, const char *);
 /* function pointer */
 p = strcmp;
 /* assign address of strcmp to p */
 printf("Enter two strings.\n");
 gets(s1);
 gets(s2);
 check(s1, s2, p);
 /* pass address of strcmp via p */
 return 0;
}
void check(char *a, char *b,
 int (*cmp)(const char *, const char *)) {
 printf("Testing for equality.\n");
 if(!(*cmp)(a, b)) printf("Equal");
 else printf("Not Equal");
}
```

Rechnerarchitektur

49

strcmp is a function in string.h

## **Strings**

- Kein „String“ Datentyp. Strings sind einfach Arrays von chars, wobei das letzte Zeichen den Wert 0 haben muss ('\0').
- Die Stringfunktionen von C verlassen sich auf dieses '\0' !
- Strings werden also über einen Zeiger (char\*) auf das erste char referenziert.
- Zeichen innerhalb eines Strings können wie in jedem anderen Array auch über s[x] oder über \*(s+x) angesprochen werden.
- Strings können auch statisch deklariert werden:  
`char *s="Dies ist ein String";`
- Warnung: Bei Allozieren (malloc) von Speicher für Strings nicht Platz für das Nullbyte vergessen. Der String „otto“ braucht also 5 Bytes!

There is no string data type

Strings are simply arrays of chars where the last char must be 0

The string functions rely on this \0!

Strings have a pointer to char to the first element

Chars can be addressed as in any other array via s[x] or \*(s+x)

Strings can also be declared to be static

Warning: when allocating (malloc) memory for strings remember to have 1 extra space for \0  
For example “otto” needs 5 bytes!

## Kommandozeilenparameter

- Kommandozeilenparameter werden auf Aufrufparameter der Funktion main gemappt:

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

- argc gibt die Anzahl der Kommandozeilenparameter an.
- „*argv ist ein Zeiger auf ein Array von Zeigern, die auf Arrays von chars (aka Strings) zeigen . .:-)*“
- **Keine Panik !:** Die einzelnen Tokens können ganz einfach mit argv[x] referenziert werden.

## Command line parms

The command line arguments are mapped to the main function as follows

Argc is the number of parms

Argv is a pointer to an array of pointers to arrays of chars (strings)

No panic! The individual elements (tokens) can be obtained via argv[x]

## argc, argv Beispiel

- Ausgabe aller Kommandozeilenparameter:

```
#include <stdio.h>

int main(int argc, char **argv)
{
 int i;
 for(i=0;i<argc;i++)
 printf("Token %d ist %s\n", i, argv[i]);
}
```

- Das erste Token (**argv[0]**) ist der Programmname selber.

Output all command line parms

The first Token argv[0] is the name of the program itself.

## Maschinennahe C-Programmierung

### ❑ Nachteile von Maschinenprogrammierung:

- schlechte Les- und Wartbarkeit

### ❑ Nachteile von Assemblerprogrammierung:

- viele Befehle zur Lösung einfacher Probleme
- geringe Portabilität

→Hochsprachen, z.B. C

### ❑ Speicherzugriffe in C

```
char *zeiger = (char *)0x1001;
*zeiger = 0x12;
```

Machine oriented C-programming

Disadvantages of machine programming:

    poor readability and maintainability

Disadv of assembly language

    many commands for solving common problems

    low portability

High-level languages such as C

    memory access in C

Speicherzugriff (memory access)

## Einbinden von Assembler-Anweisungen

- ❑ C-Spracherweiterung erlaubt Nutzung sämtlicher Möglichkeiten durch C-Programme
- ❑ Assembler-Programmierung nur an den unbedingt notwendigen Stellen
- ❑ C-Übersetzer überlässt die Verarbeitung der Assembler-Befehle einem Inline-Assembler (in C-Übersetzer integriert oder autonom)
- ❑ Übersetzung eines C-Programms in zwei Stufen
  - Erzeugen eines Assemblerprogramms durch Übertragen des eingebetteten Assembler-Codes
  - Inline-Assembler übersetzt Assembler-Programm

Integration of assembler instructions

C-language extension allows use all possibilities by C programs

Assembly language programming should only be reserved to the absolutely necessary jobs

C compiler leaves the processing of assembler commands to the inline-assembler

(built in the C compiler or independently)

For example the command in an inline assembler  
(built in C compiler)

Translation of a C program in 2 stages

generate an assembler program by transferring the embedded assembler code

inline assembler translates assembly language program

## Beispiel: Inline-Assembler

```
#define CONTROL_REGISTER 0x1001
#define RECEIVE_REGISTER 0x1002
#define ACTIVATE 0x12
#define TIMEOUT 1000
volatile unsigned char receive;
/* Variable soll im Speicher abgelegt werden, da im
 Assemblerpart darauf zugegriffen wird: _receive */
void main() {
 int time; receive = 0;
 for (time = 0; (receive != ACTIVATE) &&
 (time < TIMEOUT); time++) {
 /* Empfangenes Datum wird durch CTRL_REG erkannt
 Kopieren des Empfangsregisters in receive */
 asm{
 btst.b #4,CONTROL_REGISTER
 bne _11
 move.b RECEIVE_REGISTER,_receive
 _11:
 }
}
```

Rechnerarchitektur

55

Example:

Variable should be stored in memory as in the accessed assembler part it is:  
`_receive`

Received data is recognized by `CTRL_REG`  
copy the received register to `receive`

Flik, S. 225

## Assemblermodule

- ❑ Entwickeln von verschiedenen Modulen in C und Assembler
- ❑ Binden der Module zu einem Programm
- ❑ Aufrufen der Assembler-Unterprogramme durch C-Unterprogramme
- ❑ C- und Assembler-Unterprogramme müssen gleichen Konventionen (Parameterübergabe, Rücksprung) gehorchen.
- ❑ C-Übersetzer gibt Konventionen zur Parameterübergabe vor.

Assembler modules

Develop from different modules in C and assembler

Integrate the modules into a program

Call the assembler subroutines

By C subroutines

C and assembler subroutines must follow the conventions (parameter passing, return values)

C compiler specifies the conventions for parameter passing.