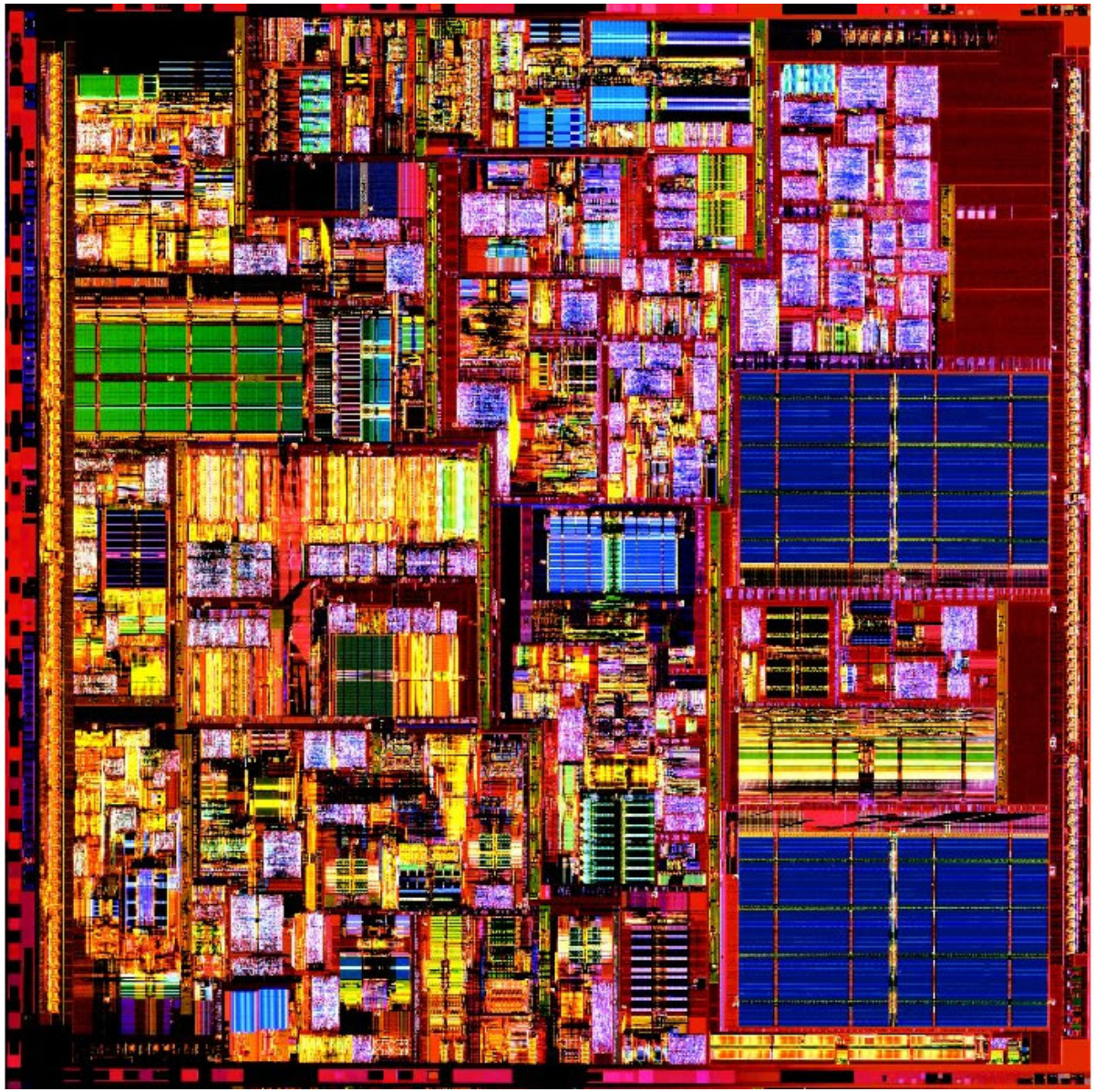
Rechnerarchitektur



Kontakt

Dr. Markus Anwander

Email: markus.anwander@inf.unibe.ch

WWW: http://cvg.unibe.ch

Fragen via https://piazza.com/unibe.ch/spring2022/2419/home

Termine

- Vorlesung: dienstags von 13.00-15.00 Uhr
- Übungen: dienstags von 15.00-16.00 Uhr
- Sprechstunde: nach Vereinbarung (email schicken)
- Fragen via https://piazza.com/unibe.ch/spring2022/2419/home
- Assistenten
  - Sari Alp Eren (TA)
  - Sepehr Sameni (TA)
  - Abdelhak Lemkhenter (TA)
  - Salomon Bruelisauer (HA)

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:
  - 1978: Erstes Buch über die Programmiersprache C:
    “The C Programming Language” bzw. “Programmieren in C”.
  - 1988 Zweitausgabe: Aktualisiert nach ANSI-C Standard.
- 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
```

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
```

#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;
  int 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+=step;
  }
}
```

Programmstruktur

```
#include <stdio.h>

#define L_LIMIT 0
#define U_LIMIT 300

int pepe;       /* globale Variable */

main()
{
  int fahr, celsius;
  int 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+=step;
  }
}
```

Includes & Defines

globale Variablen
globale Deklarationen
(structs, typedefs)

Funktionsdeklarationen

Funktionen

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
- char <= short <= int <= long
- Typ sagt wie Bitmuster im Speicher interpretiert werden muss

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 */
```

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 formatierte Eingabe !

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

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 is 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 */
```

Operatoren (Übersicht)

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

? - 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); /* !!!! 4 6*/

  /* z.B. Arrayelemente auf 0 setzen */
  for(i=0;i<10;x[i++]=0);
}
```

Kontrollstrukturen

- for(...;...;...){...;}
- do{...;}while(...);
- while(...){...;}
- if(...){...;} else {...;}
- break, continue

```
int i=0;

while(i<10){
  printf("%d\n",i);
  ++i;
}

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

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

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;
}
```

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“);
}
```

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);
}
```

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;
```

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;};
```

Typedef

- „Umbenennen“ eines Variablentyps
- 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;
  }
  ```
  ```

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.

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

```
01 #include <stdio.h>
02
03 int f(int z)
04 {
05     return z*z;
06 }
07
08 int main()
09 {
10     printf("%d\n",f(17));
11 }
```

- Funktionsdeklaration

```
01 #include <stdio.h>
02
03 int f(int z);
04
05 int main()
06 {
07     printf("%d\n",f(17));
08 }
09
10 int f(int z)
11 {
12     return z*z;
13 }
```

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	1002		65			

```
int x;       // 32-Bit
int *z_x;    // 16-Bit
x = 65;
z_x = &x;
```

Der Wert der Variablen x ist in der Speicherzelle 1002 abgelegt

Der Wert der Variablen z_x ist in der Zelle 1000 gespeichert.

z_x enthält die Adresse des Wertes von x.

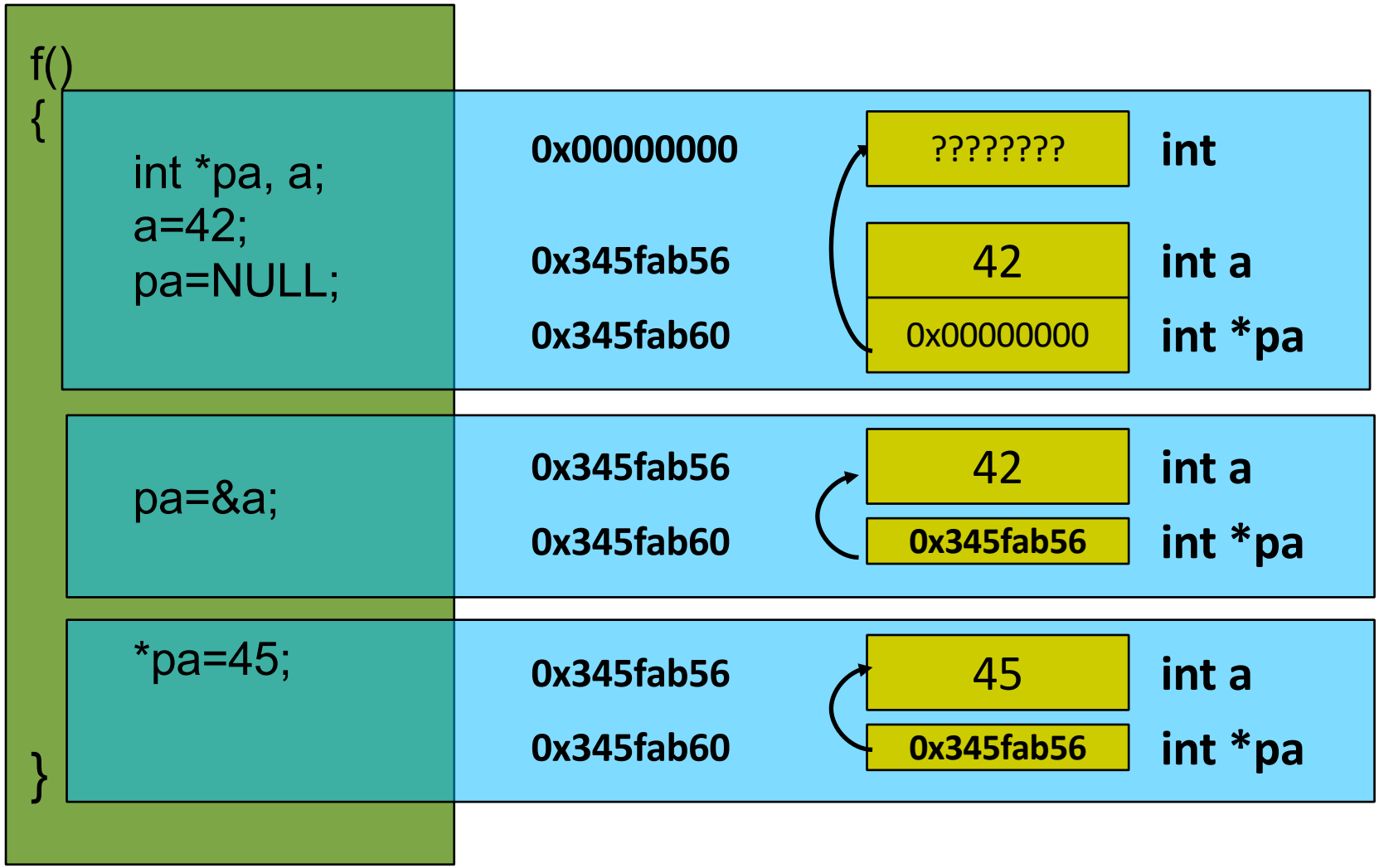
Pointer

- 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 */
     printf("%p %p\n", pa, pb);             /* Adressen */
}
```

Pointer im Schema

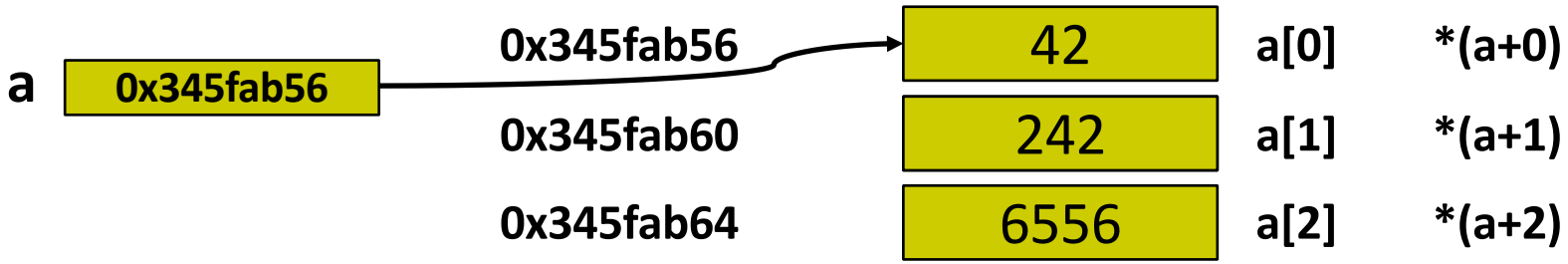


Pointer und Arrays

- 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.**



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";  // 12 * 1 Byte
        char *p;

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

- **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.**

Pointer Arithmetik (Beispiele)

```
main(){
        double v[20];  // (double 8 Byte)
        double *p;

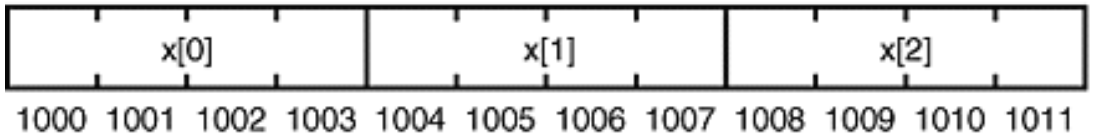
        for(p=v;p-v<20;p++)
                printf("%f\n",*p);

        for(p=v;
            (unsigned long)p-(unsigned long)v<20*sizeof(double);
            p++)
                printf("%f\n",*p);
}
```

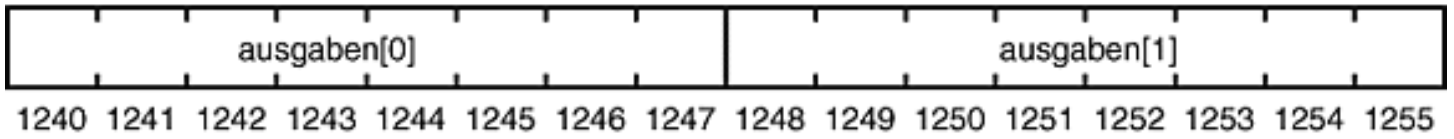
- p++ erhöht Zahlenwert von p um sizeof(double) und nicht um 1!
- Ergebnis der Subtraktion p-v ist also in sizeof(double) Einheiten.
- Gilt auch für alle anderen arithmetischen Operatoren

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 Fortsetzung

```
main(){                                          Output:
      int x, y;
      x = 10;
      printf ("x = %d\n",x);                     x = 10
      *((&y)+1) = 20;
      printf ("x = %d\n",x);                     x = 20
}
```

y und x sind auf hintereinanderfolgenden Speicherplätzen abgelegt.

Pointer Arithmetik Fortsetzung

```
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

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)
```

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)\n",(*p).x,(*p).y);  // lang
        printf("(%d/%d)\n",p->x,p->y);      // kurz
        return 0;
  }
  ```

- ```
  int f(struct Point p)
  {
        printf("(%d/%d)\n",p.x.p.y);
        return 0;
  }
  ```

- Gilt natürlich auch für bitfelder und unions !

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“.

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.

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.

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.

Dynamischer Speicher: Pitfall

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

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

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

- 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!**

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.

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\n");

      else printf("Not Equal\n");
}
```

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!

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.

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.

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;
```

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

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
  Assemblerteil 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 _l1
       move.b RECEIVE_REGISTER,_receive
  _l1:
}}}
```

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.