

System-Programmierung

1: Erste Schritte in C

CC BY-SA 4.0, T. Amberg, FHNW
(Soweit nicht anders vermerkt)

Slides: tmb.gr/syspr-1



[Source](#) auf GDocs

Überblick

Diese Lektion behandelt die *Basics* der Sprache C.

Vieles ist relativ ähnlich wie in Java.

Neu sind *Pointer* (Zeiger).

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

hello.c

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

```
$ nano hello.c {Text einfügen} CTRL-X Y ENTER
$ gcc -o hello hello.c
$ ./hello
hello, world
```

C

Entstanden 1970 an den Bell Labs, auf **UNIX / PDP-11**.

Entwickelt von Dennis Ritchie aus Vorgänger **B, BCPL**.

Standardisiert als **C89** (auch ANSI C), und später **C99**.

C im Vergleich mit Java

Die Sprache C ist prozedural, nicht* Objekt-orientiert.

Manuelle Speicherverwaltung, kein Garbage Collector.

Maschinen-nah, weniger Typ-sicher, explizite Pointers.

Variablen, Konstanten, Zuweisung

Integer Variablen, Initialisierung:

```
int b; int i, j; int k = 0;
```

Integer Konstante mit *const*:

```
const int a = 42;
```

Zuweisung (Assignment):

```
b = a;
```

Symbolische Konstanten

Definition symbolischer Konstanten mit `#define`:

```
#define PI 3.14159
```

Jedes Auftreten der Konstante wird textuell ersetzt:

```
f = PI * r^2; // =>  
f = 3.14159 * r^2;
```

`#defines` werden ohne ; und GROSS geschrieben.

Integer Typen

Deklaration von Integer (Ganzzahl) Typen:

```
char c; // Grösse sizeof(char) = 1 Byte  
int i; // Hardware-abhängig  $N \geq 2$  Byte  
short s; // bzw. short int s;  $N_{\text{int}} \geq N \geq 2$  Byte  
long l; // bzw. long int l;  $N \geq N_{\text{int}}$   
long long m; // bzw. long long int m;  $N \geq N_{\text{long}}$ 
```

Alle davon auch *unsigned*, ohne Vorzeichen:

```
unsigned int i; // Wertebereich 0 ..  $2^{N*8}-1$   
int i; // Wertebereich  $-2^{N*8-1}$  ..  $2^{N*8-1}-1$ 
```

Floating Point Typen & Wertebereich

Deklaration von Floating Point (Gleitkomma) Typen:

```
float f; // sizeof(float) ist HW-abhangig  
double d; // sizeof(double) ist HW-abhangig  
long double ld; // sizeof(...) ist HW-abhangig
```

Hardware bzw. Compiler-abhangige Konstanten:

```
#include <limits.h> // fur int Typen  
#include <float.h> // fur float Typen
```

Hands-on, 10': `sizeof()` Operator

Schreibe ein Programm, das die Größen in Bytes der Basistypen *char*, *int*, *long*, *float*, *double* ausgibt.

Nutze dazu den `sizeof()` Operator und `printf()`:

```
$ ./my_sizeof
sizeof(char) = 1
sizeof(int) = ...
```

Die Ausgabe von *int* Werten ist möglich mit `%d`, z.B.:

```
printf("%d\n", i); // \n = newline character
```

Boolean

C89 hat keinen eingebauten Boolean Typ - aber z.B.:

```
typedef enum { FALSE, TRUE } Boolean; // [TLPI]  
Boolean b = TRUE; // bzw. FALSE
```



Achtung: in

Expressions

z.B. if() gilt

alles != 0

als true.

C99 hat einen *bool* Typ in *stdbool.h*:

```
#include <stdbool.h>  
bool b = true; // bzw. false
```

Aber oft wird einfach *int* verwendet:

```
int b = 1; // bzw. 0
```

Formatierung

Formatierte Ausgabe mit *printf*:

```
printf("%c", c); // char c
printf("%d", i); // int i
printf("%ld", l); // long l
printf("%f", f); // float f
printf("%3.f", f); // 3 Vorkommastellen
printf("%.2f", f); // 2 Nachkommastellen
printf("%s", b ? "true" : "false"); // bool b
```

Expressions

Expression (Ausdruck) vom Typ *int*:

```
int a, b;  
a = 1 + 2 * 3; // Punkt vor Strich  
b = 6 * a; // b = 6 * (1 + (2 * 3))
```

Expression vom Typ *float*:

```
float c, d, e, f;  
c = b * 0.25; // int * float => float  
d = c - e - f; // (c - e) - f => v.l.n.r
```

Auswertungsreihenfolge & -richtung

$()$	$f(x)$	$[]$	$->$.	v.l.n.r.	\wedge		v.l.n.r.			
$! \sim ++ -- + - * \&$					v.r.n.l.	$ $		v.l.n.r.			
$(type)$	sizeof					$\&\&$		v.l.n.r.			
$*$	$/$	$\%$			v.l.n.r.	$ $		v.l.n.r.			
$+$	$-$	binär, a+b			v.l.n.r.	$?:$		v.r.n.l.			
$<<$	$>>$				v.l.n.r.	$=$	$+ =$	$- =$	$/ =$	$\% =$	v.r.n.l.
$<$	$<=$	$>$	$>=$		v.l.n.r.	$\& =$	$\wedge =$	$ =$	$<< =$		
$==$	$!=$				v.l.n.r.	$>> =$					
$\&$	binär, a&b				v.l.n.r.	,			v.l.n.r.		

Typumwandlung

Implizite Umwandlung, bei Zuweisung, *float* zu *int*:

```
int i = 2.3; // .3 fällt weg
```

Explizite Umwandlung, mit Typecast, *int* zu *float*:

```
float f = (float) i;
```

Typumwandlung in Operationen

Integral Promotion und arithmetische Umwandlung:

{char, short} → int → unsigned int → long →
unsigned long → long long → float → double →
long double

Bei gemischten Operationen gewinnt "grösster" Typ:

```
int i = 42;  
float f = 23.0;  
i + f; // hat Typ float, weil int → ... → float
```

Kontrollfluss

Bedingte Ausführung mit *if*:

if (*condition*) *statement*

```
if (a < 0) { a = 0; }
```

Bedingte Ausführung mit *if* und *else*:

if (*condition*) *statement₁*, **else** *statement₂*

```
if (a < b) { c = a; } else { c = b; }
```

Bedingte Ausführung mit (mehreren) *else if*:

```
if (condition1) statement1, else if (condition2)  
statement2 else statement3
```

```
if (result >= 0) {  
    printf("Success\n");  
} else if (result == -1) {  
    printf("Error No. 1\n");  
} else {  
    printf("Unknown error\n");  
}
```

Bedingte Ausführung mit *switch*:

[switch.c](#)

```
switch (expression) {  
    case const-expression: statement1,  
    default: statement2  
}
```

```
int ch = getchar();  
switch (ch) {  
    case 'y': result = 1; break;  
    case 'n': result = 0; break;  
    default: result = -1;  
}
```

Wiederholung mit *while*-Schleife:

while (*condition*) *statement*

```
int i = 0;  
while (i < 7) {  
    printf("%d\n", i);  
    i = i + 1;  
}
```

Wiederholung mit *for*-Schleife:

for (*init-expr*; *condition*; *loop-expr*) *statement*

```
for (int i = 0; i < 7; i++) {  
    printf("%d\n", i);  
}
```

```
int i = 0; // init-expr  
while (i < 7) {  
    printf("%d\n", i);  
    i++; // loop-expr  
}
```

Wiederholung mit *do-while*-Schleife:

[do_while.c](#)

do statement while (condition)

```
int c;  
do {  
    printf("enter a number [0-9]: ");  
    c = getchar();  
} while (c < '0' || '9' < c);
```

Sprung zum Ende des Blocks mit *break*-Statement:

break;

```
0: while (1) {  
1:     break; // springt zu Zeile 3  
2: }  
3:
```

Sparsam verwenden, oder mit *switch* zusammen.

Sprung zur nächsten Iteration mit *continue*:

continue;

```
0: int i = 0;  
1: while (i < 3) {  
2:     continue; // springt zu Zeile 1  
3:     i++;  
4: }  
5:
```

Sparsam oder gar nicht verwenden.

Beliebige Sprünge mit *goto*-Statement:

goto label;

...

label: *statement*

Nicht verwenden, führt zu absolut unlesbarem Code.

E. W. Dijkstra: "Go-to statement considered harmful".

Arrays

arrays.c

Deklaration eines *float* Arrays mit 3 Elementen:

```
float values[3];
```

Deklaration und Initialisierung eines Arrays:

```
float zeros[3] = { 0 }; // alle Elemente = 0
float values[] = { 20.1, 23, 15.2 };
```

Lesen / Schreiben einzelner Array-Elemente:

```
float f = values[i]; // (0 <= i) && (i < 3)
values[2] = 7.0;
```

Pointers

pointers.c

Ein Pointer (Zeiger) ist eine Variable, welche die Speicheradresse einer anderen Variable enthält:

```
int *p; // p = Pointer auf int Variable
```

Adressoperator &:

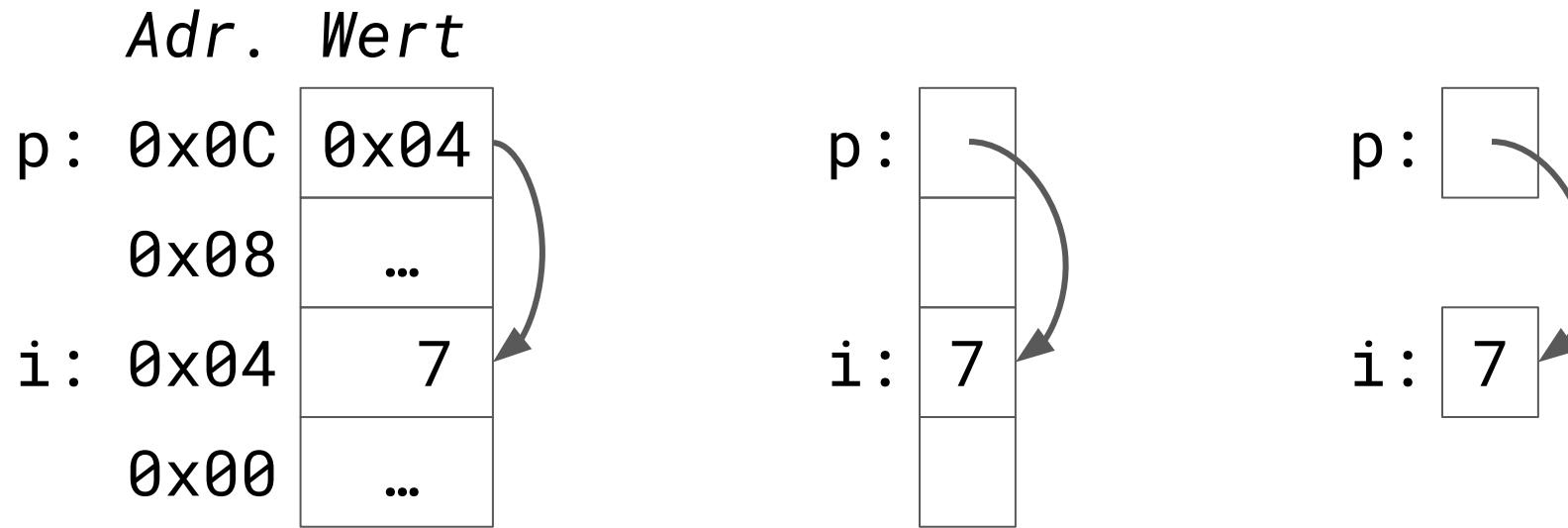
```
p = &i; // p = Adresse von i => p zeigt auf i
```

Dereferenzierungsoperator *:

```
int j = *p; // j = Wert auf den p zeigt
```

Speichermodell

Fortlaufend adressierte Speicherzellen, in jeder Zelle kann ein Wert stehen, z.B. eine Zahl oder Adresse.



Null-Pointer

C garantiert, dass θ keine gültige Speicheradresse ist:

```
char *p = 0; // Zuweisung von 0 ist erlaubt
```

stdio.h definiert die symbolische Konstante **NULL**:

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

```
char *p = NULL; // Lesbarer als 0
```

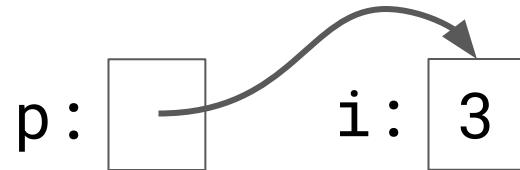
Pointer und Zahlen $\neq 0$ sind nicht austauschbar:

```
char *p = 7; // Fehler
```

Wert ersetzen, auf den ein Pointer zeigt

Dereferenzierungsoperator kann auch links stehen:

```
int i = 7; // int Variable mit Wert 7
int *p; // Pointer auf int Variable
p = &i; // p = Adresse von i => p zeigt auf i
*p = 3; // Wert an der Stelle auf die p zeigt
printf("%d", i); // => i hat jetzt den Wert 3
```



Adressarithmetik

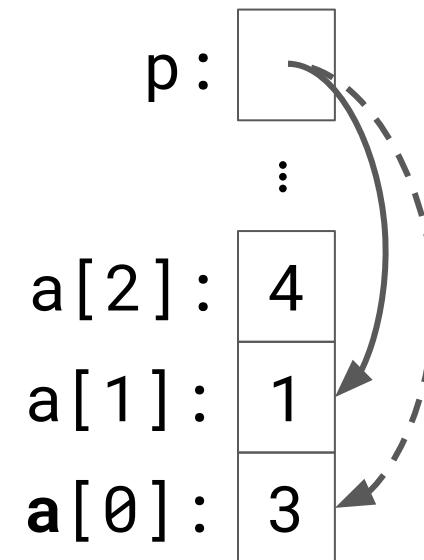
arr_ptr.c

Pointers und Arrays:

```
int a[ ] = { 3, 1, 4 };  
int *p;  
p = &a[0]; // p zeigt auf a[0]  
p = p + 1; // +1 * sizeof(int)  
int b = *p; // Wert von a[1]
```

Kurzschreibweise:

```
p = a; // bedeutet p = &a[0]
```



Strings

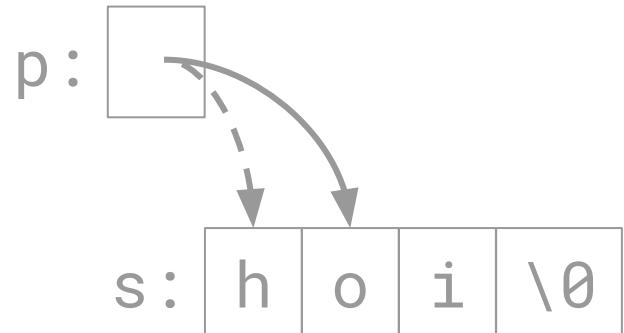
strings.c

Strings sind Arrays von *char*, mit '\0' terminiert:

```
char s[ ] = {'h', 'o', 'i', '\0'};  
for (char *p = s; *p != '\0'; p++) {  
    printf("%c", *p);  
}  
p: 
```

Oder:

```
char *s = "hoi";  
printf("%s", s);
```



String Funktionen

Die *string.h* Library enthält Standard-Funktionen.

Länge des Strings *s*, bzw. Index des ersten '\0' in *s*:

```
int strlen(const char *s);
```

Kopieren von *src* nach *dest*, Pointer auf *dest* zurück:

```
char *strcpy(char *dest, const char *src);
```

Anhängen von *src* an *dest*, Pointer auf *dest* zurück:

```
char *strcat(char *dest, const char *src);
```

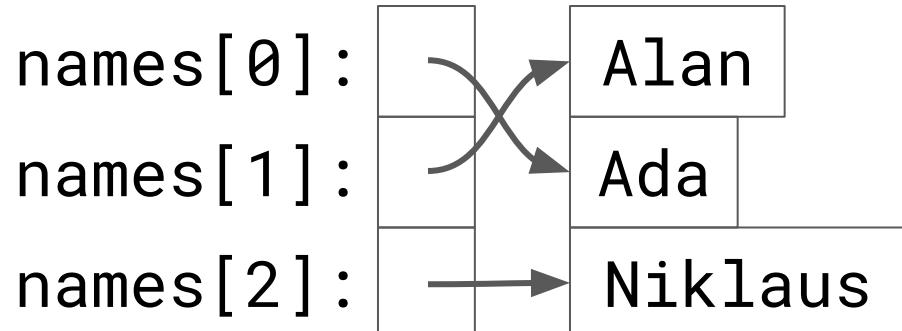
Array von Strings bzw. Pointers

Strings können in einem Array enthalten sein:

```
char *names[ ] = { "Ada", "Alan", "Niklaus" };
```

Hier ist *names* ein Array von Pointers auf Strings.

Die Strings selbst liegen irgendwo im Speicher.



Hands-on, 15': Argumente lesen args.!c

Das System übergibt Command-Line Argumente so:

```
int main(int argc, char *argv[]) { ... }
```

Schreibe ein Programm *args.c*, das seine

Argumente, d.h. alle Strings im Array *argv* ausgibt:

```
$ ./args hoi ...
```

```
0: ./args, 1: hoi, ...
```

Erweitere das Programm so, dass es einen Fehler ausgibt, falls ein Argument nicht aus [a-z]* besteht.

Mehrdimensionale Arrays

2-D Matrix von 3×4 *int* Werten:

```
int m[3][4] = { // 3-er Array von 4-er Arrays
    {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 7, 0, 0}
};

int x = m[2][1]; // nicht m[2,1]; x = 7;
```

Unterschied zu Pointers:

```
int m[3][4]; // 12 int-grosse Speicherzellen
int *n[3]; // 3 Pointer, nicht initialisiert
```

Structs

structs.c

Struct-Typ namens *point* mit *int* Feldern *x* und *y*:

```
struct point { int x; int y; };
```

Deklaration einer Variable vom Struct-Typ *point*:

```
struct point p = { 3, 2 };
```

```
struct point q; // immer mit struct keyword
```

Zugriff auf Struct Felder mit Punkt-Notation:

```
q.x = p.y;
```

Pointer auf Structs

Pointer auf Struct-Typ namens *point*:

```
struct point *p;
```

Zugriff auf Struct Feld erfordert Klammern:

```
(*p).x; // damit * vor . ausgewertet wird
```

Dasselbe geht aber auch kürzer, mit -> Notation:

```
p->x;
```

Typen definieren mit *typedef* *typedef.c*

Typ namens *Point* mit *int* Feldern *x* und *y*:

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

Deklaration einer Variable vom Typ *Point*:

```
Point p = { 3, 2 };
```

```
Point q; // ohne struct keyword
```

Struct-Typen können geschachtelt werden:

```
typedef struct rect { Point a; Point b; } Rect
```

Speicher allozieren

Speicher auf dem Stack allozieren, zur Compile-Zeit:

```
Point ps; // alloziert Speicher auf dem Stack  
Point *p = &ps; // p zeigt auf Adresse von ps
```

Speicher auf dem Heap allozieren mit *malloc*:

```
Point *p = malloc(sizeof(Point));
```

Heap-Speicher freigeben mit *free*:

```
free(p); // manuell, kein Garbage Collector
```

Hands-on, 15': Bäume

tree.!c, _v2.!c

Erstelle eine Datei *tree.c* mit einem Typ *struct node* mit Zeigern auf *left*, *right* vom selben Typ, und einem String *label* von maximal 32 Byte Länge.

Instanziere einen binären Baum mit 3 Blättern, verwende dazu die Funktionen *malloc* und *free*.

Empfohlene Compiler Flags

Für eine möglichst strikte Analyse im *gcc* Compiler:

```
$ gcc my.c
-std=c99 // oder -std=c89 (auch -ansi)
-pedantic // Strikte ISO C Warnungen
-pedantic-errors // Strikte ISO C Errors
-Werror // Behandle Warnungen als Errors
-Wall // Einschalten "aller" Warnungen
-Wextra // Einschalten von extra Warnungen
```

Programme bauen mit *make*

makefile

Einfaches *makefile*

```
$ cd fhnw-syspr/01  
$ cat makefile
```

...

Builden (bauen) mit *make*

\$ make all	Alle Programme bauen
\$ make hello	Einzelnes Programm bauen
\$ make clean	Erzeugte Programme löschen

Hands-on, 15': Makefile

Erstelle ein makefile für Ihren Hands-on Code.

Verwende die Compiler Flags aus dem Script.

Korrigiere allfällige neue Kompilationsfehler.

Führe *make clean* aus, vor dem *git commit*.

Selbststudium: Grundlagen in C

Zur Vertiefung, lese folgende Kapitel in [K&R]

4: Functions & Program Structure bis p.88.

5: Pointers and Arrays

6: Structures

Feedback oder Fragen?

Gerne in Teams, oder per Email an

thomas.amberg@fhnw.ch

Danke für Ihre Zeit.

Dr. Donna Malayeri comma PhD

https://twitter.com/lindydonna/status/1044705793885388800

Home

Follow

Tweet

RT if you learned to program before StackOverflow existed. I'm jealous of how much easier it is to learn now!

11:50 PM - 25 Sep 2018

205 Retweets 199 Likes

38 205 199

Dan Selman @danielselman · Sep 26

Replying to @lindydonna @tedneward

I remember using a dial-up BB to ask questions about programming in C... I thought that was pretty amazing compared to ordering books and waiting for Dr Dobbs. These youngsters have no idea! ;-)

2 6

Dr. Donna Malayeri comma PhD @lindydonna · Sep 26

Also explains why my first instinct is to read the docs, whereas they just Google.