System-Programmierung 1: Erste Schritte in C

n|w

Ablauf heute

⅓ Vorlesung,

⅔ Hands-on,

Feedback.

Slides, Code & Hands-on: tmb.gr/syspr-1

hello.c

```c
#include <stdio.h>

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
```

n|w

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.

n|w

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.

n|w

Variablen, Konstanten, Zuweisung

Integer Variablen, Initialisierung:

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

Integer Konstante mit *const*:

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

Zuweisung (Assignment):

```c
b = a; // b = 42
a = b; // Fehler
```

n|w

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.

n|w

Integer Typen

Deklaration von Integer (Ganzzahl) Typen:

```
char c; // Grösse sizeof(char) = 1 Byte
int i; // Hardware-abhängig N ≥ 4 Byte
long l; // bzw. long int l; N ≥ 4 Byte ≥ N_int
short s; // bzw. short int s; N_int ≥ N ≥ 2 Byte
```

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
```

n|w

Hands-on, 15': *int* Wertebereich range.!c

Hardware bzw. Compiler-abhängige Konstanten:

```
#include <limits.h>
```

Schreiben Sie ein Programm *my_range.c*, das für die Typen *char, int, long, short* Wertebereiche so ausgibt:

```
type: TYPE_SIZE byte, TYPE_MIN .. TYPE_MAX
```

Erweitern Sie das Programm für *unsigned* Typen*.

(Tipp: `$ gcc FILE.c -M` zeigt *include* Pfade an.)

n|w

Floating Point Typen & Wertebereich

Deklaration von Floating Point (Gleitkomma) Typen:

```
float f; // sizeof(float) ist HW-abhängig
double d; // sizeof(double) ist HW-abhängig
long double ld; // sizeof(…) ist HW-abhängig
```

Hardware bzw. Compiler-abhängige Konstanten:

```
#include <float.h>
```

Interne Darstellung meistens* IEEE 754.

n|w

Boolean

Bei C89 gibt es keinen eingebauten Boolean Typ:

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

Achtung: in Expressions z.B. if( ) gilt alles != 0 als true.

Bei C99 gibt es den *bool* Typ in *stdbool.h*:

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

n|w

Formatierung

Formatierung mit *printf*:

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

n|w

Expressions

expr.c

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
```

n|w

Auswertungsreihenfolge & -richtung

( ) f(x) [ ] -> .	v.l.n.r.	^	v.l.n.r.
! ~ ++ -- + - * &	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.	&= ^= |= <<=	
== !=	v.l.n.r.	>>=	
& binär, a&b	v.l.n.r.	,	v.l.n.r.

n|w

Typkonversion

upper.c

Implizit, bei Zuweisung:

```
int i = 2.3; // .3 fällt weg
```

Explizit, mit Typ-cast:

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

Integer Promotion und arithmetische Konversion:

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

n|w

Kontrollfluss

Bedingte Ausführung mit *if*:

```
if (condition) statement
```

Bedingte Ausführung mit *if* und *else*:

```
if (condition) statement1 else statement2
```

Empfehlung: Statements jeweils mit Block *{ }*, z.B.

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

n|w

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");
}
```

n|w

Bedingte Ausführung mit (mehreren) ~~*else*~~ *if*:

```
int animal = PLATYPUS;
if (lays_eggs(animal)) {
    …
} else if (is_mammal(animal)) {
    …
} else {
    …
}
```

n|w

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;
}
```

n|w

Wiederholung mit *while*-Schleife:

```
while (condition) statement
```

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

n|w

Wiederholung mit *for*-Schleife:

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

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

n|w

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);
```

n|w

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.

n|w

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.

n|w

Beliebige Sprünge mit *goto*-Statement:

```
goto label;
…
label: statement
```

Nicht verwenden, führt zu absolut unlesbarem Code.

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

n|w

Arrays

arrays.c

Deklaration eines *float* Arrays mit 3 Elementen:

```
float temp_values[3];
```

Deklaration und Initialisierung eines Arrays:

```
float temp_values[3] = { 20.1, 23, 15.2 };
```

Lesen / Schreiben einzelner Array-Elemente:

```
t = temp_values[i]; // (0 <= i) && (i < 3)
temp_values[2] = 7.0;
```

n|w

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 *:

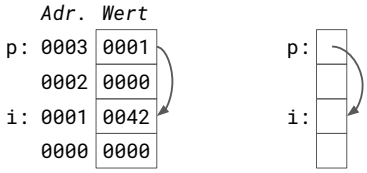
```
j = *p; // j = Wert auf den p zeigt => i
```

n|w

Speichermodell (stark vereinfacht)

Fortlaufend adressierte Speicherzellen.

In jeder Zelle kann ein Wert stehen.



n|w

Null-Pointer

C garantiert, dass *0* 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 bloss 0
```

Pointer und Zahlen *!= 0* sind nicht austauschbar:

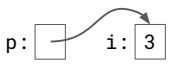
```
char *p = 7; // Fehler
```

n|w

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
```



n|w

Adressarithmetik

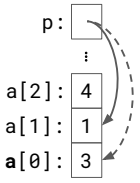
arr_ptr.c

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]
```



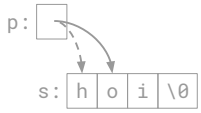
n|w

Strings

strings.c

Strings sind Arrays von *char*, mit Null terminiert:

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



Oder:

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

n|w

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);
```

n|w

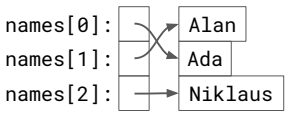
Pointer Arrays

Pointer sind Variablen, können in Arrays drin sein:

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

Sortieren wird so effizienter, ändert nur Pointer:

```
qsort(names, 0, 2, …);
```



n|w

Hands-on, 15': Argumente lesen

args.!c

Command-Line Argumente als Parameter von *main*:

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

Schreiben Sie ein Programm *my_args.c*, das alle Command-Line Argumente mit Index ausgibt:

```
$ ./my_args hoi => 0: ./my_args, 1: hoi
```

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

n|w

Mehrdimensionale Arrays

2-D Matrix von 3 x 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
```

n|w

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;
```

n|w

Pointer auf Structs

Pointer auf Struct-Typ namens *point*:

```
struct point *p;
```

Zugriff auf Struct Feld erfordert Klammern:

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

Dasselbe geht deshalb auch kürzer, mit -> Notation:

```
p->x;
```

n|w

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
```

n|w

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
```

n|w

Hands-on, 15': Bäume

tree.!c, _v2.!c

Erstellen Sie eine Datei *my_tree.c* mit einem Struct Typ *Node* mit Zeigern auf *left*, *right* vom selben Typ, und einem String *label* von maximal 32 Byte Länge.

Instanzieren Sie einen binären Baum mit 3 Blättern, verwenden Sie dazu die Funktionen *malloc* und *free*.

Erweitern Sie den *Node* Typ für Bäume mit variabler Anzahl (N > 2) Ästen, wie Darwin's Tree of Life.

n|w

Unions als "Variant Record"

variants.c

Ein Union-Typ nimmt einen von mehreren Typen an:

```
union variant { // union variant ist
    int int_value; // entweder ein int
    float float_value; // oder ein float
} v; // sizeof(v) gross genug für grössten Typ
```

Auch mit *typedef*, wie bei *struct*:

```
typedef union variant { … } Variant;
Variant v = { .float_value = 23.0 };
```

n|w

Unions und Bit-weiser Zugriff unions.c

Mehrere Typen als "Sicht" auf dieselbe Speicherstelle:

```
typedef union packet {
    unsigned char byte_value;
    struct bit_layout {
        unsigned int lo_nibble : 4;
        unsigned int hi_nibble_lsb : 1;
    } bit_values; // Hardware-abhängig!
} Packet;         // $ lscpu => Little Endian
```

n|w

Hands-on, 15': BLE Pakete ble.!c

Erstellen Sie ein C *Struct* Typ für BLE Pakete gemäss: https://devzone.nordicsemi.com/f/nordic-q-a/12211/ble-packet-structure in einer neuen Datei *my_ble.c*

n|w

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
```

n|w

Programme builden mit *make* makefile

Einfaches *makefile*

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

Builden (bauen) mit *make*

```
$ make all        Alle Programme builden
$ make hello      Einzelnes Programm builden
$ make clean      Erzeugte Programme löschen
```

n|w

Hands-on, 15': Makefile

Erstellen Sie ein makefile für Ihren Hands-on Code.

Verwenden Sie die Compiler Flags aus dem Script.

Korrigieren Sie allfällige neue Kompilationsfehler.

Führen Sie *make clean* aus, vor dem *git commit*.

n|w

Selbststudium, 3h: Functions & Structure

Als Vorbereitung auf die nächste Lektion, lesen Sie [K&R] *4: Functions & Program Structure* bis p.88.

Die nächste Lektion fasst den Lesestoff zusammen, ohne Selbststudium wird das Tempo eher hoch sein.

n|w

Feedback?

Gerne im Slack oder an thomas.amberg@fhnw.ch

Programmierfragen am besten schriftlich.

Sprechstunde auf Voranmeldung.

Slides, Code & Hands-on: tmb.gr/syspr-1