

System-Programmierung

2: Funktionen

CC BY-SA, Thomas Amberg, FHNW
(soweit nicht anders vermerkt)

Ablauf heute

$\frac{1}{3}$ Vorlesung,

$\frac{2}{3}$ Hands-on,

Feedback.

Slides & Hands-on: tmb.gr/syspr-2



Funktionen in C

Definition einer Funktion:

```
return-type function-name(parameter-decl's) {  
    declarations and statements  
}
```

```
int max(int a, int b) {  
    int m;  
    if (a > b) { m = a; } else { m = b; }  
    return m;  
}
```

Deklaration einer Funktion:

```
return-type function-name(parameter-decl's);
```

```
int max(int a, int b);
```

Aufruf einer Funktion:

```
function-name(arguments);
```

```
m = max(5, 7);
```

```
n = max(max(6, PI), 7);
```

```
printf("%d", max(3, 4));
```

```
max(5, 7); // ignoriert Resultat
```

Argumentübergabe "by value"

power.c

Parameter *base* und *n* sind Kopien der Argumente:

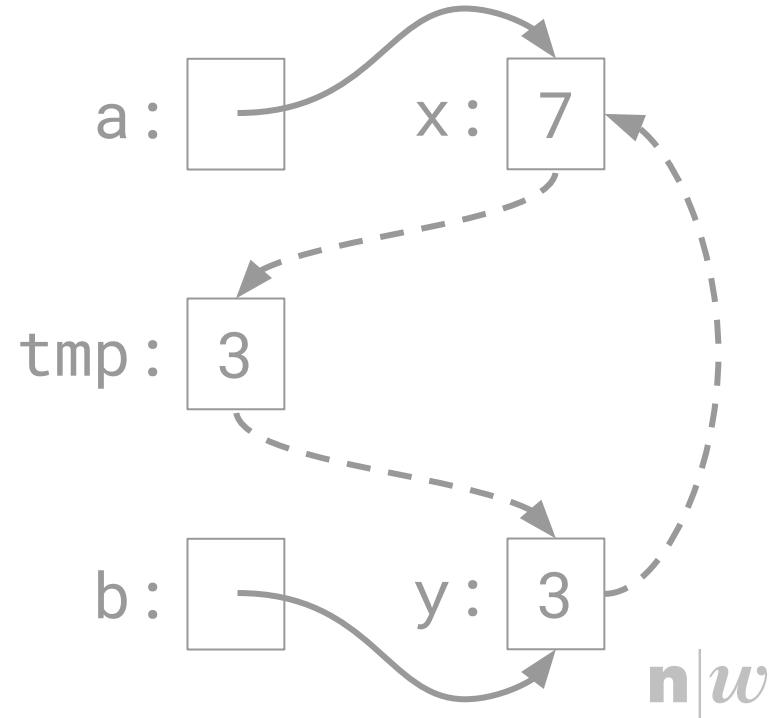
```
int power(int base, int n) {  
    int result = 1;  
    while (n > 0) {  
        result *= base;  
        n--;  
    }  
    return result;  
}
```

Argumentübergabe "by reference" `swap.c`

Parameter / Argumente zeigen auf dasselbe:

```
void swap(int *a, int *b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

```
int x = 3, y = 7;  
swap(&x, &y);
```



Funktion in Datei auslagern main.c, f.c

Deklaration der Funktion *f* in *main.c*:

```
void f(void); // nur f() hiesse  $\geq 0$  Argumente
```

C Dateien einzeln mit *gcc -c* kompilieren:

```
$ gcc -c f.c           erzeugt f.o
```

```
$ gcc -c main.c        erzeugt main.o
```

Ein Programm aus den Objektdaten linkern:

```
$ gcc -o my_program main.o f.o
```

Basis-Typen zurückgeben

z.B. Return-Wert vom Typ *float*:

```
float parse_float(char s[]) { ... return f; }
```

Deklaration mit *float* Return-Wert:

```
float parse_float(char s[]);
```

Aufruf ist ein Ausdruck vom Typ *float*:

```
int parse_int(char s[]) {  
    return (int) parse_float(s);  
}
```


Struct-Typen zurückgeben

struct.c

z.B. Return-Wert vom Typ *Point*:

```
typedef struct { int x; int y; } Point;  
Point create_point(int x, int y) { ... }
```

Deklaration mit *Point* Return-Wert:

```
Point create_point(int x, int y);
```

Aufruf ist ein Ausdruck vom Typ *Point*:

```
Point origin = create_point(0, 0);
```

Hands-on, 20': Heap Struct `struct_v2.c`

In `struct.c` wird ein Struct auf dem Stack alloziert, mit `return` zurückgegeben und dabei "by value" kopiert.

Schreiben Sie ein Programm `my_struct_v2.c`, das für `create_struct` Pointer und `malloc` verwendet:

```
Point *create_point(int x, int y);
```

Passen Sie den restlichen Code entsprechend an, der Compiler gibt Ihnen dabei nützliche Hinweise.

Globale Variablen

count.c

Globale, "externe" Variable bleibt erhalten:

```
int count; // global

void f() { count++; }

int main() {
    f(); f(); f();
    // count = 3
}
```

Scope Regeln

scope.c

Der Scope einer Variable beginnt mit der Deklaration:

```
int b = a; // error: a undeclared
int a = 0;
int b = a; // ok: a was declared
```

Dieselbe Regel gilt für die Sichtbarkeit in Funktionen:

```
void f() { int j = i; } // error: i undeclared
int i;
```

Ein lokaler Scope endet am Ende des `{ }` Blocks.

Funktionen sollten im Voraus deklariert werden:

```
void f() { g(); } // warning: implicit decl.  
void g() { ... } // (error, falls gcc -Wall)
```

Deklaration von *g()* ohne *void* wäre nicht korrekt:

```
g(); // default return Typ ist int [K&R p.30]
```

Falls Reihenfolge fix, hilft Vorwärts-Deklaration:

```
void g(void); // forward declaration  
void f() { g(); }  
void g() { ... }
```

Um Variablen in mehr als einer Datei zu benutzen, werden sie einmal definiert, und mehrfach deklariert:

```
int i; // Definition von i und (unten) Array a,  
int a[32]; // Speicher wird auf Stack alloziert
```

Deklaration einer Variable, die extern definiert ist:

```
external int i; // Deklaration von i und a[],  
external int a[]; // kein Speicher alloziert
```

```
external int i = 0; // Fehler, nicht erlaubt  
external int a[32]; // Dimension ist optional
```

Header Dateien

/nest

Eine Header Datei erlaubt, Deklarationen zu teilen:

```
// nest.h
```

```
#define MIN_TEMP 5  
void nest_up(void);  
void nest_down(void);  
int nest_temp(void);
```

```
// nest.c
```

```
#include "nest.h"  
int temp; ...
```

```
// home.c
```

```
#include "nest.h"  
void home_leave() {  
    nest_down(); ...  
}
```

Statische Variablen

static.c

Variablen sind über Dateigrenzen hinweg sichtbar:

```
int temp; // in nest.c, sichtbar in home.c
```

Modifizier *static* begrenzt Sichtbarkeit auf die Datei:

```
static int temp; // nur sichtbar in nest.c
```

In Funktionen beschränkt *static* den Scope auf diese.

Der Zustand bleibt über Funktionsaufrufe hinweg da:

```
void f() { static int count = 0; count++; }
```


Block Struktur

scope.c

Jeder Block `{ }` spannt einen eigenen Scope auf:

```
int i = 0; // "extern", globaler Scope
void f() { // nicht geschachtelt
    int i = 1;
    { // freistehender Block
        int i = 2;
    }
    if (...) { int i = 3; ... } else { ... };
}
```

Initialisierung

garbage.c

Globale, "externe" und *static* Variablen sind Null:

```
int i; // per Default mit 0 initialisiert  
char c = '0' + 3; // konstante Expression
```

Lokale, "automatische" Variablen sind undefiniert:

```
void f() {  
    int i; // = Garbage(!)  
}
```

Compiler Flags können hier helfen, siehe [makefile](#). **n|w**

Rekursion

fib.c

Eine Funktion kann sich selbst aufrufen:

```
int f(int n) {  
    if (n < 2) { // "Abbruchbedingung"  
        return n;  
    } else {  
        return (f(n-1) + f(n-2)); // Rekursion  
    }  
}
```

Pointers auf Funktionen

map.c

Funktion *map*, die Funktionen auf Arrays anwendet:

```
void map(int a[], int len, int (*f)(int));  
int inc(int i); // Beispiel-Funktion
```

Implementierung wendet *f* auf die Elemente von *a* an:

```
for (int i=0; i<len; i++) { a[i] = f(a[i]); }
```

Aufruf mit *f = inc* Funktion, die ein *int* inkrementiert:

```
map({0, 0, 7}, 3, inc); // => {1, 1, 8}
```

Präprozessor

Jedes *#include* wird mit dem Datei-Inhalt ersetzt:

```
#include "file-name" // sucht im Source Dir.  
#include <file-name> // folgt Such-Heuristik
```

Jedes Auftreten des Tokens wird textuell ersetzt:

```
#define token-name replacement-text  
#define PI 3.14159  
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Der Scope eines Makros reicht bis zum Dateiende. **n|w**

Hands-on, 5': Makros (auf Papier) max.!c

Gegeben ein Makro:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Und (separat) die Funktion:

```
int max(int a, int b) { return a > b ? a : b; }
```

Was passiert beim folgenden Aufruf?

```
int i = 1, j = 0;
```

```
int m = max(++i, j); // mMakro = ?, mFunktion = ?
```

Präprozessor #if:

#if int-expression

#elif int-expression

#else

#endif

(z.B. \$ cat /usr/include/assert.h)

Bedingte #defines:

#ifndef token-name (oder #ifdef token-name)

#define token-name

#endif

Kompilationsprozess

Schritt für Schritt:

```
$ echo "int main() {}" > my.c
```

```
$ cpp my.c > my.i           => my.i
```

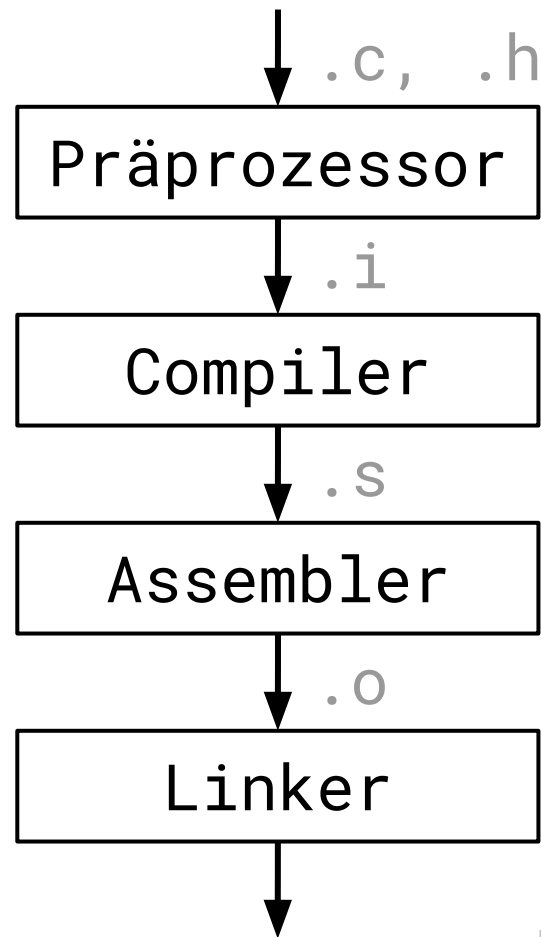
```
$ gcc -S my.i               => my.s
```

```
$ as -o my.o my.s          => my.o
```

```
$ ld -o my my.o ...        => my
```

Was *gcc* wirklich macht:

```
$ gcc -v -o my my.c
```



Hands-on, 20': Kompilationsprozess

Kompilieren Sie schrittweise mit *cpp*, *gcc*, *as* und *ld*.

Sehen Sie sich den Inhalt des jeweiligen Outputs an.

Versuchen Sie, den *ld* Schritt zum Laufen zu bringen.

Objekt- bzw. Binärdateien können Sie anzeigen mit :

```
$ objdump -x my.o
```

```
$ hexdump -C my
```

Libraries

Eine Library (Programmbibliothek) besteht aus vor-kompilierten Objektdateien die mit einem Linker in ein Programm gelinkt werden können.

Statische Libraries *.a* werden ins Programm kopiert.

Dynamische Libraries *.so* werden zur Laufzeit in das Programm gelinkt, mit "dynamic linking". Der Code kann von mehreren Programmen genutzt werden.

System-Programmierung

Neben der Programmiersprache C brauchen wir für System-Programmierung ein Verständnis des UNIX/Linux Betriebssystems, das in den Modulen *bsys* und *sysad* ausführlich behandelt wurde.

Betriebssystem-Kern

Betriebssystem kann auch Tools bedeuten, hier eher Core OS, *Kernel*; verwaltet Linux System-Ressourcen.

Prozess-Scheduling; Memory-Management; Dateisystem; Prozesse starten / beenden; Device-Zugang verwalten (USB etc.); Networking; *System Call API*.

Kernel- und User-Mode

Die CPU läuft im *Kernel-Mode* oder im *User-Mode*.

Teile des virtuellen Speichers können als User- bzw. Kernel-Space markiert werden; User dürfen weniger.

Manche Operationen sind nur dem Kernel erlaubt:
z.B. der Zugang zur Speicherverwaltungs-Hardware,
die Instruktion *halt* und Operationen für Geräte-I/O.

Kernel- und Prozess-Sicht

Für ein Prozess passieren Dinge asynchron, er weiss nicht, wann und wie lange er die CPU für sich hat, ob er im RAM oder ausgelagert ist, und wo auf der Disk Dateien physisch abgelegt sind; wie Device I/O geht.

Der Kernel macht das alles transparent für Prozesse.

Ein Prozess "kreiert einen Prozess" heisst eigentlich er "bittet den Kernel einen Prozess zu kreieren".

System Calls

Ein *System Call* ist ein kontrollierter Eintrittspunkt in den Kernel, der seine Dienste via *API* bereitstellt.

Bei System Calls geht die CPU in den Kernel-Mode.

Argumente werden kopiert v. User- zu Kernel-Space.

Jeder System Call hat einen Namen und eine Nr./ID.

Siehe [syscalls.h](#) in Linux, [syscallent.h](#) in *strace*.

Standard / GNU C Library

version.c

Standard C Bibliothek auf Linux Systemen ist **glibc**:

```
$ ldd file | grep libc => /libc.so.6
```

```
$ /lib/arm-linux-gnueabi/libc.so.6 => v2.24
```

Version kann auch per Programm bestimmt werden:

```
printf("%d, %d", __GLIBC__, __GLIBC_MINOR__);
```

```
#include <gnu/libc-version.h>
```

```
const char *gnu_get_libc_version(void);
```

```
printf("%s", gnu_get_libc_version());
```


Error Handling

error.c

Fail fast! System Call Fehlercodes immer behandeln.

Viele System Calls geben im Fehlerfall -1 zurück, und der Fehlercode steht in der globalen Variable *errno*:

```
#include <errno.h>
fd = open(pathname, flags, mode);
if (fd == -1) { // Fehlerbehandlung
    if (errno == EINTR) { ... } else { ... }
}
```

Achtung: Erfolgreiche Calls setzen errno nicht auf Null.

Fehlermeldung ausgeben mit *perror()*:

errno.c

```
#include <stdio.h>  
perror("open"); // liest errno
```

Oder Meldung mit *strerror()*:

```
#include <string.h>  
char *msg = strerror(errno);
```

Manche System Calls geben im *Erfolgsfall* -1 zurück;
dort setzt man *errno* vor dem Aufruf auf 0.

System-Datentypen

sys_t.c

Die Grösse von *int*, *long*, etc. ist Hardware-, und die Grösse von System-Datentypen Versions-abhängig, deshalb werden Standard C Typen verwendet, z.B.:

```
#include <sys/types.h> // Definiert pid_t, ...  
typedef int pid_t; // Typ für Prozess IDs  
pid_t pid = ...; // Code wird portabler
```

Durch *typedef* wird die "Implementierung" der PID Grösse von der Verwendung im Code entkoppelt.

Standard C Typ *pid_t* ist auf ARM Linux so definiert:

```
$ cat /usr/include/arm-linux-gnueabi/\
/sys/types.h | grep pid_t
typedef __pid_t pid_t;
$ cat /usr/includ.../bits/types.h | grep __pid_t
__STD_TYPE __PID_T_TYPE __pid_t;
$ cat /u.../bits/typesizes.h | grep __PID_T_TYPE
#define __PID_T_TYPE __S32_TYPE
$ cat /usr/inc.../bits/types.h | grep __S32_TYPE
#define __S32_TYPE int
```

Hands-on, 30': Kilo.c

Analysieren Sie den Source Code dieses Programms:

<https://github.com/antirez/kilo/blob/master/kilo.c>

Kompilieren Sie das Programm und benutzen Sie es.

Was macht das Programm? Gibt es extra Features?

Was fällt Ihnen im Source Code besonders auf?

(@antirez ist auch der Autor von [Redis](#).)

Selbststudium, 3h: File In-/Output

Als Vorbereitung auf die nächste Lektion, lesen Sie [\[TLPI\]](#) *Chapter 4: File I/O, The Universal I/O Model*.

Das [PDF des Kapitels 4](#) ist verfügbar als Teil der offiziellen "Downloadable samples from the book".

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

Feedback?

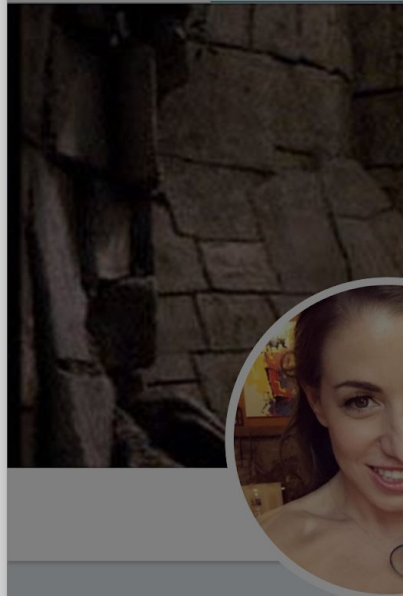
Gerne im [Slack](#) oder an thomas.amberg@fhnw.ch

Programmierfragen am besten schriftlich.

Sprechstunde auf Voranmeldung :)

Slides, Code & Hands-on: tmb.gr/syspr-2





Sarah Drasner

@sarah_edo

Award-winning speaker
@vuejs Core Team
organizer @Conc
codepen.io/sdras
she/her



Sarah Drasner

@sarah_edo

Follow

Me: guess what?
Dizzy: what?
Me: recursion is fun! Guess what else?
Dizzy: what?
Me: recursion is fun! Guess what else?
Dizzy: boundary condition.

This is why I married him. ❤️

8:36 PM - 12 Sep 2018

294 Retweets 2,829 Likes



39



294



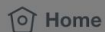
2.8K



Tweet your reply



Max Bougie @monkchips · Sep 13



Max Bougie



@monkchips

Following

Replying to @sarah_edo



Home [bell icon]



@selfsame@tiny.tilde.website
@jplur_
computers were a
Delray Beach,



@selfsame@tiny.tilde.website
@jplur_

Following [dropdown arrow]

The recursive centaur: half horse, half recursive centaur



9:12 PM - 11 Sep 2018

7,842 Retweets 32,467 Likes



236 7.8K 32K [reply icon] [retweet icon] [like icon] [direct message icon]

Tweet [close icon]

