Betriebssysteme (BS)

Einführung in C

https://sys.cs.tu-dortmund.de/de/lehre/ss23/bs

Peter Ulbrich

peter.ulbrich@tu-dortmund.de
bs-problems@ls12.cs.tu-dortmund.de

Basierend auf *Betriebssysteme* von Olaf Spinczyk, Universität Osnabrück

technische universität
dortmund

arbeitsgruppe
systemsoftware

Programmierparadigmen

- **imperative** Programmierung
  - Programm: Folge von Befehlen
- **prozedurale** Programmierung
  - Spezialfall der imperativen Programmierung
  - Programm: Menge von Prozeduren, die auf gemeinsamem Datenbestand operieren
- **objektorientierte** Programmierung
  - Kapselung von Code und Daten in Objekten
  - Programm: Menge von Objekten, die über Schnittstellen interagieren

  → aber i.d.R.: „OO Sprache“ = imperativ + OO-Erweiterung

- Ganz anders: **deklarative** Programmierung (funktional, regelorientiert ...)

Programmierparadigmen (2)

- Eine Sprache kann **für ein bestimmtes Paradigma besonders geeignet** sein, erzwingt jedoch nicht dessen Verwendung oder verbietet die Verwendung anderer Paradigmen.
- Beispiele:
  - prozedurale Programmierung in Java (*god object*, *big hairy object*)
  - nichtprozedurale imperative Programmierung in C (*god function*)
  - objektorientierte Programmierung in C

Java vs. C

- hello_world.java

```java
class Hello {
    public static void main(String argv[]) {
        System.out.println("Hello world!");
    }
};
```

- hello_world.c

```c
#include <stdio.h>

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

  - `printf()` ist nicht Teil der Sprache, sondern der Standardbibliothek
  - `main()` gehört zu keiner Klasse (u.a. weil es in C keine Klassen gibt)
  - `main()` benötigt einen Rückgabewert: den exit-code des Programms

Struktur von C-Programmen

```c
#include <stdio.h>

int counter = 0;

int gcd(int a, int b) {
    counter++;
    if (a == 0) return b;
    if (b == 0) return a;
    return gcd(b, a % b);
}

int main(void) {
    int eastwood = 10164;
    char guevara = 240;
    printf("result: %d\n", gcd(guevara, eastwood) );
    printf("function calls needed: %d\n", counter);
    return 0;
}
```

globale Variablendefinitionen und Funktionen

lokale Variablen am Anfang einer Funktion/eines Blocks

- main()-Funktion
  - Einsprungpunkt für jedes C-Programm
  - Rückgabewert bildet den **Exit-Code** des Programms

Ausgaben mit printf

```c
#include <stdio.h>

int main(void) {
    int eastwood = 4711;
    printf("Eine Zahl: %d\nUnd jetzt hexadezimal: %x", -815, eastwood);
    return 0;
}
```

- notwendig, damit der Compiler `printf` „kennt“:
  `#include <stdio.h>`
- erster Parameter: **Formatstring**
  `"Eine Zahl: %d\nUnd jetzt hexadezimal: %x"`
- enthält **Platzhalter** für weitere Parameter:
  - dezimal mit (eventuellem) Vorzeichen: %d (vorzeichenlos: %u)
  - hexadezimal: %x
  - viele weitere in der Manpage zu `printf` (siehe Übung)

Funktionen

- „Klassenfreie Methoden“
- Elementare Bausteine für die **Modularisierung** imperativer Programme
  - **Verringerung der Komplexität** durch Zerteilen komplexer Probleme in überschaubare Teilaufgaben
  - **wiederverwendbare** Programmkomponenten
  - **Verbergen** von Implementierungsdetails
- Funktionen vs. Methoden
  - werden global deklariert und definiert
  - sind nicht Teil einer Klasse
  - kennen daher kein `this`

Funktionen – Deklaration/Definition

- Funktionen sollten deklariert werden, bevor sie aufgerufen werden.

```
void bar(int);              /* Deklaration */

void foo(int b) {
    if (b < 0) return;
    bar(b - 1);
}

void bar(int a) {           /* Definition */
    if (a < 0) return;
    foo(a - 1);
}
```

- ***forward*-Deklaration** sorgt dafür, dass Compiler `bar` bereits „kennt“, wenn er `foo` übersetzt
  - ansonsten Annahme, dass Rückgabewert vom Typ `int` ist (implizite Deklaration), und abgeschaltete Parameter-Typüberprüfung
    → schlechter Programmierstil, erzeugt Compilerwarnung
  - (historischer Hintergrund: *One-pass compiler*)

Funktionen – Werteaustausch

- Java
  - einfache Datentypen: *call by value*
  - Objekttypen: *call by reference**
- C
  - technisch ausschließlich *call by value*
  - (*call by reference* ist konzeptionell auch möglich: Stichwort „Zeiger“)

```c
#include <stdio.h>

void foo(int a) {
    a++;
}

int main(void) {
    int a = 5;
    foo(a);
    printf("%d", a);
    return 0;
}
```

* Eigentlich verwendet Java für Objekttypen *call by value where the value is a reference*, was nur konzeptionell *call by reference* entspricht. Siehe auch: https://stackoverflow.com/a/40523

Funktionen – Werteaustausch

- Java
  - einfache Datentypen: *call by value*
  - Objekttypen: *call by reference**
- C
  - technisch ausschließlich *call by value*
  - (*call by reference* ist konzeptionell auch möglich: Stichwort „Zeiger“)

```c
#include <stdio.h>

void foo(int a) {
    a++;
}

int main(void) {
    int a = 5;
    foo(a);
    printf("%d", a);
    return 0;
}
```

Was gibt dieses Programm aus?

* Eigentlich verwendet Java für Objekttypen *call by value where the value is a reference*, was nur konzeptionell *call by reference* entspricht. Siehe auch: https://stackoverflow.com/a/40523

Kontrollstrukturen

- funktionieren in C genau wie in Java
- `if (Bedingung) {...} else {...}`
- `while (Bedingung) {...}`
- `do {...} while (Bedingung);`
- `for(...;Bedingung;...) {...}`
- `switch (...) {case ...: ...}`
- `continue; break;`
- einziger Unterschied: Bedingung muss ganze Zahl sein (anstatt `boolean`)

Standardtypen

- Wie in Java gibt es die einfachen Datentypen
  - `char` Zeichen (ASCII-Code), 8 Bit
  - `int` ganze Zahl, (typ. 32 Bit, architekturabhängig)
  - `float/double` Gleitkommazahl (32 Bit / 64 Bit)
  - `void` ohne Wert
- zusätzlich gibt es noch die Modifikatoren:
  - `signed`, `unsigned`, `short` und `long`
  - also z.B. `long int` oder kurz: `long`
    (Java: `short` und `long` sind eigene Datentypen mit standardisierter Bitanzahl)
- den Typ `boolean` gibt es (ab C99): `bool`
  - boolesche Ausdrücke evaluieren zu 0 (falsch) oder zu 1 (wahr)
  - ganze Zahlen können wie boolsche Variablen verwendet werden

```
printf("%d", 4711 > 42);  /* gibt 1 aus */
while (1) {}              /* Endlosschleife */
```

Strukturen (structs)

- In C gibt es keine Klassen
- … wohl aber *komplexe* Datentypen (structs)
- „Klassen ohne Methoden“

```
struct student {
    int matrikelnummer;
    int alter;
    char name[64];
};

void rejuvenate(struct student s) {
    s.alter = 0;
}

void foo(void) {
    struct student s1;
    s1.alter = 20;
    rejuvenate(s1);
}
```

- struct-Parameter werden ebenfalls *by value* übergeben!

Strukturen (structs)

- In C gibt es keine Klassen
- … wohl aber *komplexe* Datentypen (structs)
- „Klassen ohne Methoden“

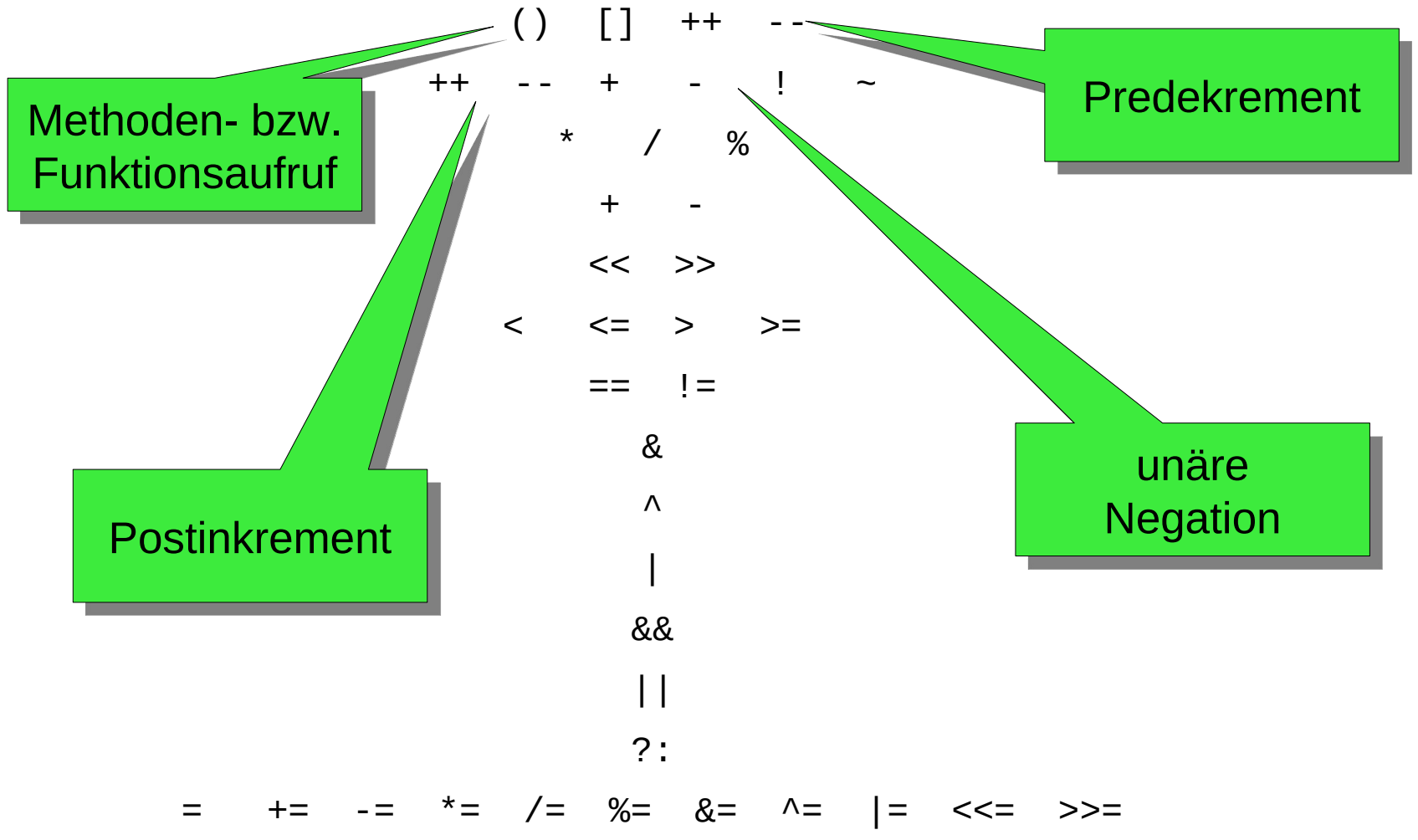
```
struct student {
    int matrikelnummer;
    int alter;
    char name[64];
};

void rejuvenate(struct student s) {
    s.alter = 0;
}

void foo(void) {
    struct student s1;
    s1.alter = 20;
    rejuvenate(s1);
}
```

Wie alt ist s1 an dieser Stelle?

- struct-Parameter werden ebenfalls *by value* übergeben!

Operatoren (1)

- haben in C größtenteils dieselbe Funktion wie in Java



Operatoren (2)

- es gibt allerdings auch Unterschiede

```
.        ()  []  ++  --        ->
(t)  ++  --  +   -   !   ~   *  &  sizeof
          *   /   %
           +   -
          <<  >>
       <   <=  >   >=
          ==  !=
            &
            ^
            |
           &&
           ||
           ?:
=   +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=
```

auch in Java, aber mit anderer Semantik!

nur in C

Operatoren (3)

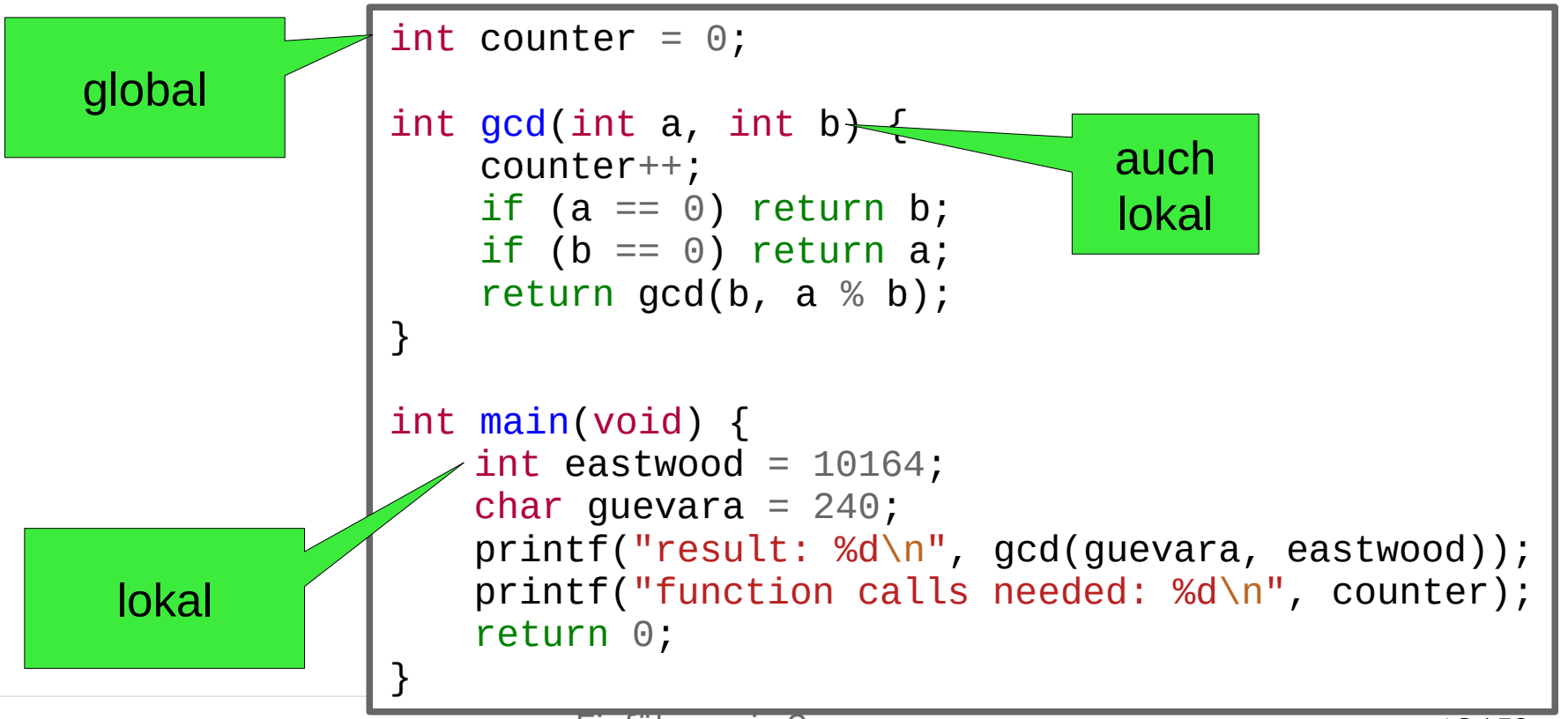
- `.` Zugriff auf Member eines Objekts
  - Zugriff auf Methode oder Membervariable eines Objekts in Java
  - In C gibt es keine Objekte mit Methoden
- `(t)` Cast nach Typ `t`
  - Wird in Java automatisch auf Gültigkeit überprüft
  - in C: Wert im Speicher wird einfach als t interpretiert

```
int eastwood = 0xD431;
char guevara = (char) eastwood;   /* guevara enthaelt dann 49, warum? */
```

- `&` Adressoperator
  - liefert die Adresse einer Variablen zurück
- `*`, `->` und `sizeof` finden später noch Erwähnung

Variablen

- müssen immer initialisiert werden (wie in Java)
  - ansonsten **undefinierter Wert**
  - Initialisierung kann direkt bei der Definition erfolgen
- können *global* oder *lokal* sein




```c
int counter = 0;

int gcd(int a, int b) {
    counter++;
    if (a == 0) return b;
    if (b == 0) return a;
    return gcd(b, a % b);
}

int main(void) {
    int eastwood = 10164;
    char guevara = 240;
    printf("result: %d\n", gcd(guevara, eastwood));
    printf("function calls needed: %d\n", counter);
    return 0;
}
```

Globale Variablen

- werden **außerhalb von Funktionen** definiert
- sind ab ihrer Definition im gesamten Programm zugreifbar
- können jedoch von lokalen Variablen *verdeckt* werden

- Probleme:
  - Zusammenhang zwischen Daten und darauf operierendem Programmcode geht verloren
  - Funktionen können Variablen ändern, ohne dass der Aufrufer damit rechnet (Seiteneffekte)
  - erschwert die Wartung von Programmen

  → globale Variablen möglichst vermeiden!

Lokale Variablen

- werden *innerhalb* und *am Anfang* einer Funktion oder eines Blocks definiert
- sind außerhalb dieser Funktion oder dieses Blocks nicht zugreifbar
- *verdecken* bei Namensgleichheit alle bisher gültigen Definitionen, welche dann innerhalb dieses Blocks nicht mehr zugreifbar sind

- Was gibt das Beispielprogramm aus?

```c
int a = 0, b = 1;

void bar(int b) {
    a = b;
}

void foo(int a, int b) {
    {
        int b = a;
        int a = 0;
        a = a + b;
    }
    bar(a);
}

int main(void) {
    int b = a;
    {
        int a = 2;
        foo(a, b);
    }
    printf("%d\n", a);
    return 0;
}
```

Zusammenfassung – bisheriger Stand

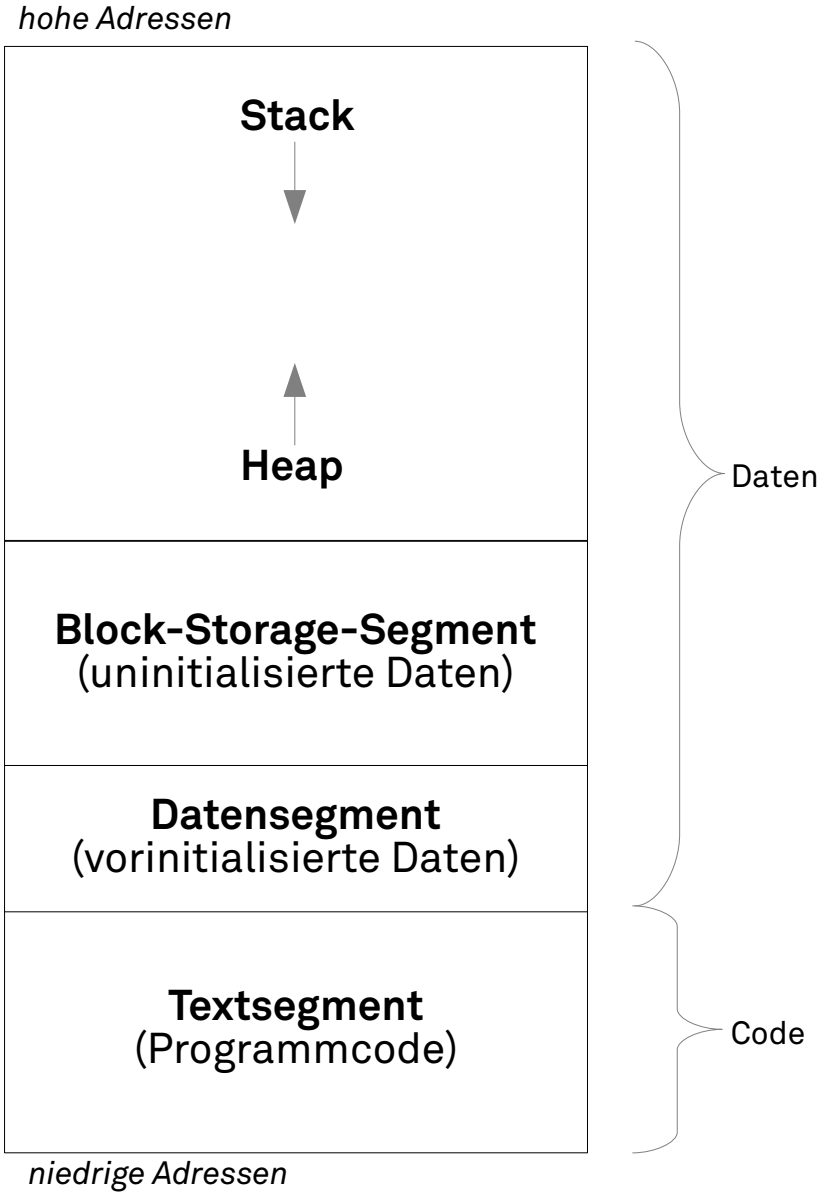
- was es nicht gibt
  - Klassen
  - Exceptions
  - `public`, `private` und `protected`
  - `new` und Garbage Collection
  - `import`
- was es sonst gibt (bisheriger Stand)
  - Funktionen
  - globale Variablen
  - `#include`

C – Schlüsselwörter

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Speicherlayout von Programmen

- **globale Variablen** landen im BSS- oder Datensegment
- **lokale Variablen** landen auf dem Stack
  - werden angelegt, wenn entsprechende Funktion betreten wird
  - gehören zum *Stackframe* einer Funktion
  - werden anschließend wieder vom Stack entfernt
- Heap: *dynamische* Variablen
  - werden in Java mit new erzeugt
  - in C: wird später behandelt (Stichworte: Zeiger, `malloc()`)



Speicherlayout (2)

mem.c

```c
#include <stdio.h>
int global_uninitialized;
int global_initialized = 4711;

void uboot(void) {}

int main(void) {
    int local;
    printf("%p\n%p\n%p\n%p\n",
        &global_uninitialized,
        &global_initialized,
        &uboot,
        &local);
    return 0;
}
```

```
js@ios:~$ gcc mem.c -o mem.elf
js@ios:~$ nm mem.elf -r -n
080495a8 A _end
080495a4 B global_uninitialized
080495a0 b completed.5843
080495a0 A _edata
080495a0 A __bss_start
0804959c D global_initialized
08049598 d p.5841
08049594 D __dso_handle
08049590 W data_start
08049590 D __data_start
08049578 d _GLOBAL_OFFSET_TABLE_
080494a4 d _DYNAMIC
080494a0 d __JCR_LIST__
080494a0 d __JCR_END__
0804949c d __DTOR_END__
08049498 d __DTOR_LIST__
08049494 d __CTOR_END__
08049490 d __init_array_start
08049490 d __init_array_end
08049490 d __CTOR_LIST__
0804848c r __FRAME_END__
0804847c R _IO_stdin_used
08048478 R _fp_hw
0804845c T _fini
08048430 t __do_global_ctors_aux
0804842a T __i686.get_pc_thunk.bx
080483d0 T __libc_csu_init
080483c0 T __libc_csu_fini
08048379 T main
08048374 T uboot
08048350 t frame_dummy
08048320 t __do_global_dtors_aux
080482f0 T _start
08048278 T _init
         U printf@@GLIBC_2.0
         U __libc_start_main@@GLIBC_2.0
         w __gmon_start__
         w _Jv_RegisterClasses
```

- Wieso ist die Variable `local` rechts nicht zu sehen?

Speicherlayout (2)

mem.c

```c
#include <stdio.h>
int global_uninitialized;
int global_initialized = 4711;

void uboot(void) {}

int main(void) {
    int local;
    printf("%p\n%p\n%p\n%p\n",
        &global_uninitialized,
        &global_initialized,
        &uboot,
        &local);
    return 0;
}
```

```
js@ios:~$ ./mem.elf
0x80495a4
0x804959c
0x8048374
0xffe62a80
```

```
js@ios:~$ gcc mem.c -o mem.elf
js@ios:~$ nm mem.elf -r -n
080495a8 A _end
080495a4 B global_uninitialized
080495a0 b completed.5843
080495a0 A _edata
080495a0 A __bss_start
0804959c D global_initialized
08049598 d p.5841
08049594 D __dso_handle
08049590 W data_start
08049590 D __data_start
08049578 d _GLOBAL_OFFSET_TABLE_
080494a4 d _DYNAMIC
080494a0 d __JCR_LIST__
080494a0 d __JCR_END__
0804949c d __DTOR_END__
08049498 d __DTOR_LIST__
08049494 d __CTOR_END__
08049490 d __init_array_start
08049490 d __init_array_end
08049490 d __CTOR_LIST__
0804848c r __FRAME_END__
0804847c R _IO_stdin_used
08048478 R _fp_hw
0804845c T _fini
08048430 t __do_global_ctors_aux
0804842a T __i686.get_pc_thunk.bx
080483d0 T __libc_csu_init
080483c0 T __libc_csu_fini
08048379 T main
08048374 T uboot
08048350 t frame_dummy
08048320 t __do_global_dtors_aux
080482f0 T _start
08048278 T _init
         U printf@@GLIBC_2.0
         U __libc_start_main@@GLIBC_2.0
         w __gmon_start__
         w _Jv_RegisterClasses
```

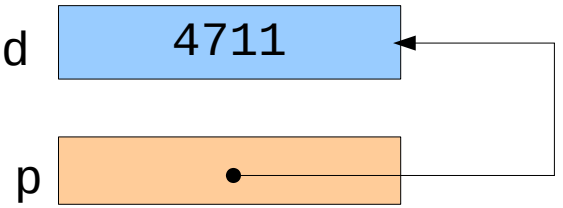
Zeiger(-variablen)

- Variablen haben eine Adresse im Speicher!
- Adresse kann wiederum in einer Variablen abgelegt werden:

```
int eastwood = 4711;
int *p;
p = &eastwood; /* p "zeigt" jetzt auf eastwood */
```

- **Variable**: Bezeichnung eines Datenobjekts

- **Zeigervariable (Pointer):**
  Bezeichnung einer Referenz auf ein Datenobjekt



Zeiger (2)

- Eine Zeigervariable (***pointer***) enthält als Wert die Adresse einer anderen Variablen: Der Zeiger zeigt auf die Variable.
- Über diese Adresse kann man **indirekt** auf die Variable zugreifen.
- große Bedeutung von Zeigern in C:
  - Funktionen können ihre Argumente ändern (bzw. das, worauf ihre Argumente *zeigen*), so macht man *Call by Reference* in C!
  - dynamische Speicherverwaltung
  - effizientere Programme
- aber auch Nachteile ...
  - Programmstruktur wird unübersichtlicher (welche Funktion kann auf welche Variablen zugreifen?)
  - häufigste Fehlerquelle in C-Programmen

Zeiger (3)

- Syntax für das Anlegen von Zeigervariablen:
  `Typ *Name;`
- Beispiel:

```
int eastwood = 4711;
int *p;
int x;

p = &eastwood; /* p "zeigt" jetzt auf eastwood */

x = *p; /* kopiert das, worauf p zeigt, nach x */
```

- Adressoperator **&** – liefert die Adresse einer Variable.
  - z.B. liefert ***`&eastwood`*** die Speicheradresse der Variable *eastwood*
- Verweisoperator ***** – ermöglicht den Zugriff auf den Inhalt einer Variable.

Zeiger als Funktionsargumente

- Parameter werden in C *by value* übergeben
- Eine Funktion kann den aktuellen Parameter beim Aufrufer **nicht verändern!**
- Auch Zeiger werden *by value* übergeben
  ⇒ Funktion erhält lediglich eine **Kopie** der Adresse.
- Über diesen Verweis kann die Funktion jedoch mit Hilfe des ***-Operators auf die zugehörige Variable zugreifen und sie verändern:
  ⇒ *call by reference*

```
void inc(int *x) {
    (*x)++;
}
int main(void) {
    int foobie = 42;
    inc(&foobie);
...
```

Zeiger als Funktionsargumente (2)

- Beispiel: Vertauschen von Variableninhalten

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

int main(void)
{
    int olerant = 42, ernational = 4711;

    swap(&olerant, &ernational);

    printf("%d %d\n", olerant, ernational);
    return 0;
}
```

- swap() erhält die Adressen der beiden Variablen
- Verwendung des *-Operators zum Zugriff auf die referenzierten Variablen

Zeiger auf Strukturen

- analog zu „Zeiger auf Variablen“
- **Beispiel:** verkettete Liste

```
#include <stdio.h> /* fuer Definition von NULL */

struct listelement {
    char guevara;
    struct listelement *next;
} a, b, c;

a.next = &b;
b.next = &c;
c.next = NULL; /* entspricht 0, Dokumentation! */
```

- **NULL**: „spezieller“ Adresswert, hier verwendet zur Terminierung der Liste

Zeiger auf Strukturen (2)

- Zugriff auf Strukturkomponenten über einen Zeiger
- bekannte Vorgehensweise:
  - *-Operator liefert die referenzierte Struktur
  - .-Operator zum Zugriff auf die Komponente
  - Operatorenvorrang: Klammerung notwendig

```
struct listelement {
    char guevara;
    struct listelement *next;
} a, *p;

p = &a;
(*p).guevara = 'a';
```

- dasselbe, nur in (syntaktisch) etwas schöner (lesbarer!):
  -> -Operator

```
p->guevara = 'a';
```

Zeiger (4)

- auch möglich, wenn auch etwas seltener im Einsatz: **Zeiger auf Zeiger** (auf Zeiger auf ...)

```
int x, *ptr, **ptr_ptr;
ptr_ptr = &ptr;
ptr = &x;
**ptr_ptr = 4711;
```

- Zeiger auf Funktionen

```
int (*func)();
func = &myfunction;
(*func)();
```

  - z.B. um einer Funktion eine andere Funktion als Parameter zu übergeben
  - **Beispiel:** Bibliotheksfunktion **`qsort()`** erhält als Parameter einen Zeiger auf eine Vergleichsfunktion, die zwei Elemente miteinander vergleicht

Typedef

- Definition eines **neuen Namens (Alias)** für existierenden Typ
- **Syntax:** wie beim Anlegen einer Variablen, **`typedef`** davor

```
typedef int Length; /* diese Zeile belegt keinen Speicher! */

Length len, max_length;
void set_length(Length l) { ... }
```

- **Abstraktionsmittel:** der tatsächlich verwendete Typ wird versteckt, kann leicht ausgetauscht werden
  (prominente Beispiele: `pid_t`, `FILE`)
- **Selbstdokumentation:** einfacher Name besser verständlich/lesbar als komplexer Pointer auf Struktur

```
typedef struct listelement *LEPtr;
LEPtr elem1, elem2;
LEPtr find_elem(LEPtr firstelem, int searchnum) { ... }
```

Felder

- aka „Arrays“
- ähnlich wie in Java, aber ...
  - Dimensionierung nur mit Konstanten! (mal abgesehen von C99...)
  - nicht initialisierte **globale** Felder sind mit 0en gefüllt
  - Inhalt von nicht initialisierten **lokalen** Feldern ist undefiniert
  - bei Initialisierung wird bei fehlenden Werte rechts mit 0en aufgefüllt

```
int primes[100] = {2, 3, 5, 7, 11, 13, 17};
/* primes[7] bis primes[99] ist 0! */

/* automatische Dimensionierung */
int even[] = {2, 4, 6, 8, 10, 12};
```

  - keine Bounds Checks beim Zugriff!
    ⇒ Effekte beim Lesen/Schreiben außerhalb der Grenzen variieren von „nichts passiert“ über Absturz des Programms bis zu völlig unvorhersagbarem Verhalten!
  - echte **mehrdimensionale Felder** sind möglich

Mehrdimensionale Felder

- Definition und Initialisierung

```
int calendar[12][31]; /* 12 “Zeilen”, 31 “Spalten” */
int uebungsgruppen_limits[][5] =
      { { 27, 27, 22, 27, 27 },   /* ungerade Wochen */
        { 27, 27, 22, 27, 27 } }; /* gerade Wochen   */
```

- Defaultwerte wie bei eindimensionalen Arrays
- Zugriff wie in Java:

```
uebungsgruppen_limits[1][3]--; /* ein Stuhl weniger */
```

Felder (2)

- Felder von Zeigern

```
int *quiesel[10];
```

- Felder von Strukturen

```
struct listelement my_elements[50] =
 { { 12, NULL }, { 80, NULL } };
```

- Felder von char
  - So werden *Strings* in C verwaltet!
  - C *Strings* sind aufeinander folgende **char**s, die mit 0 terminiert sind
  - Initialisierung wie sonst, oder aber mit Anführungszeichen

```
/* dreimal dasselbe in unterschiedlicher Syntax */
char text1[] = { 'f', 'o', 'o', 0 };
char text2[] = { 102, 111, 111, 0 };
char text3[] = "foo";
```

  - mehr zu Zeichenketten folgt später

Programme mit mehreren Dateien

main.c

```c
int main(void) {
    hello();
    return 0;
}
```

hello.c

```c
#include <stdio.h>

void hello(void) {
    printf("Hello, world!\n");
}
```

- Probieren wir's einfach mal ...

```
js@ios:~/hello$ gcc main.c hello.c -o hello_world.elf
```

- funktioniert tatsächlich:
  - Dateien werden getrennt voneinander übersetzt
  - *implizite Deklaration* von **`hello()`** (Rückgabetyp **`int`**, keine Parameter-Typüberprüfung) (warning!)
  - Compiler vermerkt hello als *undefiniertes Symbol*
  - Linker findet das Symbol in der anderen Übersetzungseinheit und versieht den Aufruf mit der richtigen Adresse
- äußerst fragwürdig (warum?)

Programme mit mehreren Dateien (2)

main.c

```c
void hello(void);

int main(void) {
    hello();
    return 0;
}
```

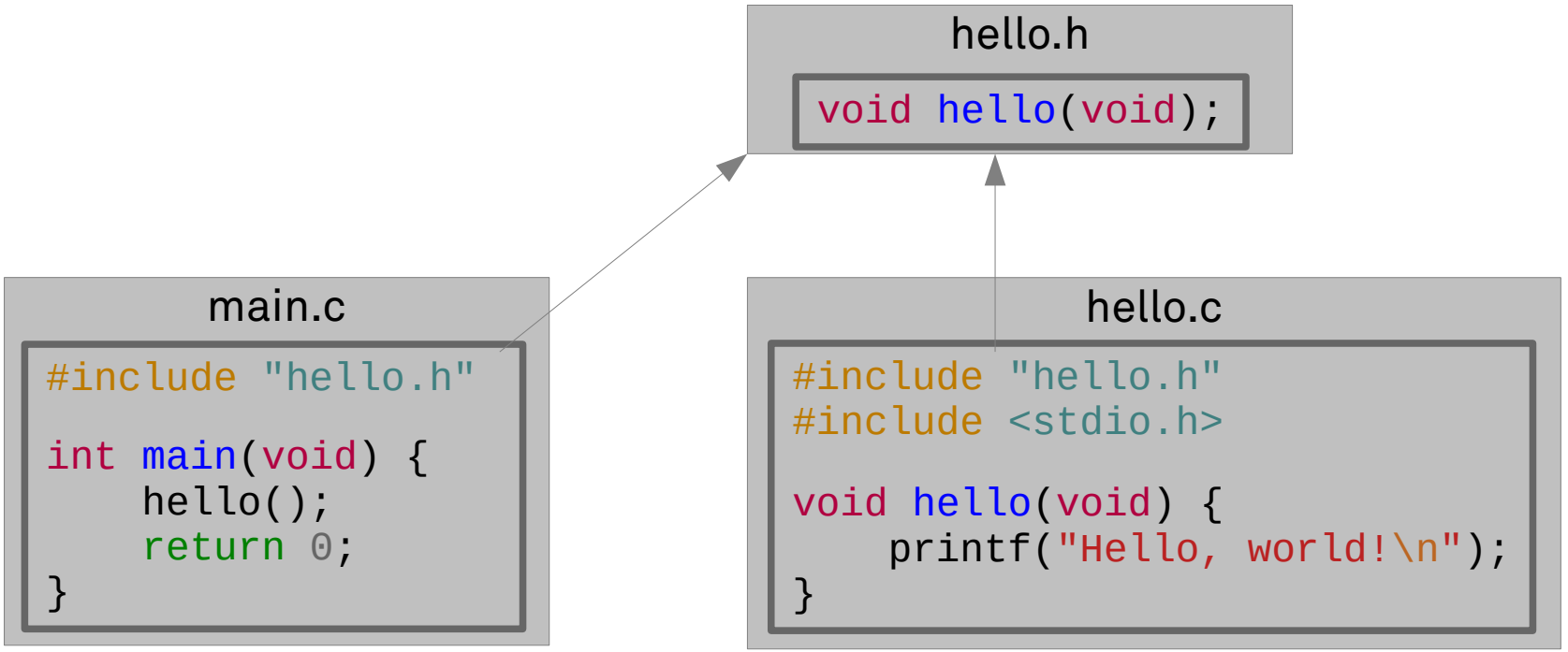
hello.c

```c
#include <stdio.h>

void hello(void) {
    printf("Hello, world!\n");
}
```

- so funktioniert's auch mit `-Werror`
- aber immer noch nicht gut
  - Deklaration von `hello()` in `hello.c` kann sich ändern!
  - Beide müssen **die gleiche Deklaration** verwenden (warum?)
- noch besser: beide verwenden **die selbe Deklaration**

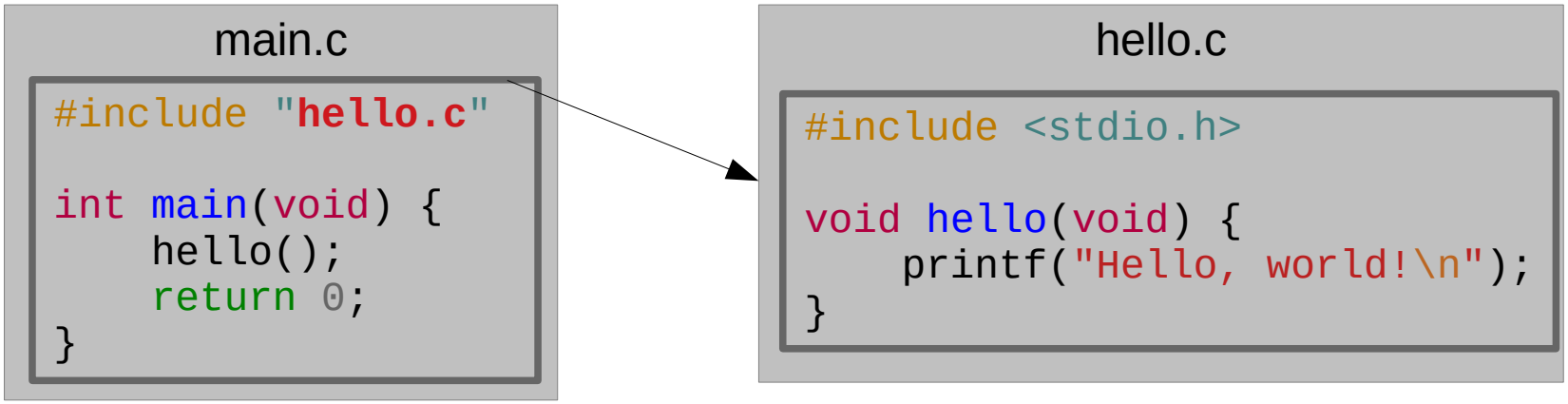
→ Nutzung des Präprozessors

Programme mit mehreren Dateien (3)



- So und nicht anders wird's gemacht!
  - `#include` ist ein *Präprozessorbefehl*
  - der Präprozessor kopiert den Inhalt der Datei an diese Stelle
  - Dateipfade in „“ sind relativ zum Verzeichnis der aktuellen .c-Datei
  - Dateipfade in <> beziehen sich compilereigene, plattformspezifische Verzeichnisse

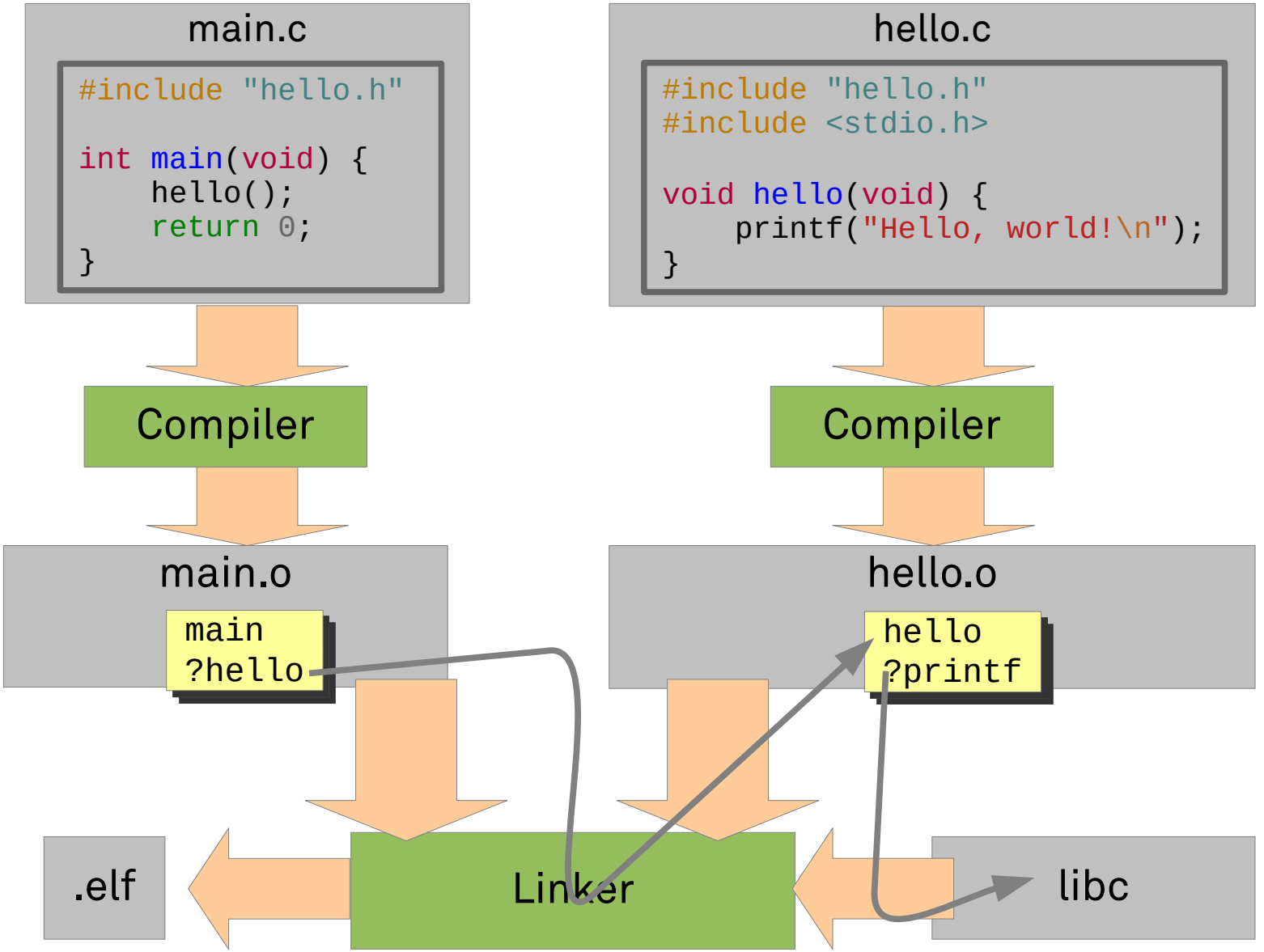
Programme mit mehreren Dateien (4)



- Vor allem nicht so!

```
js@ios:~/hello$ gcc main.c hello.c -o hello_world.elf
/tmp/ccvCawGO.o: In function `hello':
hello.c:(.text+0x0): multiple definition of `hello'
/tmp/cc6gea2y.o:main.c:(.text+0x0): first defined here
collect2: ld returned 1 exit status
```

Programme mit mehreren Dateien (5)



Module

- Bei globalen Variablen ist zu unterscheiden, ob
  - auf diese nur innerhalb eines Moduls zugegriffen werden soll.
  - oder ob auf diese auch aus anderen Modulen zugegriffen wird.
- Zugriff auf globale Variablen anderer Module mittels extern

compilation_unit_1.c

```
int eastwood;
```

compilation_unit_2.c

```
extern int eastwood;
```

- static macht globale Variablen für andere Module „unsichtbar“

compilation_unit_1.c

```
static int eastwood;
```

compilation_unit_2.c

```
extern int eastwood;
```

Linkerfehler

  - Kapselung von Daten in einem Modul
  - verhindert Namenskollisionen beim Linken
  - guter Programmierstil

Module (2)

- Um auf Funktionen anderer Module zuzugreifen ist kein `extern` notwendig...
- Funktionen können auch als `static` deklariert werden.
  - Sollte bei solchen Funktionen gemacht werden, die nur innerhalb eines Moduls verwendet werden und daher nicht zur „Modulschnittstelle“ gehören.
- Auch lokale Variablen können `static` deklariert werden.
  - allerdings mit völlig anderer Bedeutung
  - Variableninhalt „überlebt“ Funktionsaufrufe:

```
unsigned odd_number(void)
{
    static unsigned n = 1;
    return n += 2;
}
```

Präprozessor

- Definition und Benutzung von Präprozessorsymbolen

```
#define PFERD 4711
printf("%d", PFERD);
```

wird zu

```
printf("%d", 4711);
```

- bedingte Übersetzung

```
#define PFERD 4711
#ifdef PFERD
printf("Pferd.\n");
#else
printf("Kein Pferd.\n");
#endif
```

wird zu

```
printf("Pferd.\n");
```

  - Präprozessor schlau genug, Zeichenketten unverändert zu lassen
  - Zu sehr viel mehr reicht es jedoch nicht ...

```
#define PFERD 17+4
if (3*PFERD == 63) {
    ...
}
```

wird zu

```
if (3*17+4 == 63) {
    ...
}
```

Präprozessor (2)

- parametrisierte Makros

```
#define SQUARE(x) x*x
printf(“%d”, SQUARE(3));
```

wird zu

```
printf("%d", 3*3);
```

  - kein Leerzeichen zwischen Name und `(`, kein `;` am Zeilenende!
  - Fallstrick Klammerung

```
#define SQUARE(x) x*x
printf("%d", SQUARE(1+2));
```

wird zu

```
printf("%d", 1+2*1+2);
```

  - Lösung: jede Verwendung des Parameters (und auch den Gesamtausdruck) klammern!

```
#define SQUARE(x) ((x)*(x))
printf("%d", SQUARE(1+2));
```

wird zu

```
printf("%d", ((1+2)*(1+2)));
```

  - Makros über mehrere Zeilen: `\` vor dem Zeilenende

```
#define SQUARE(x) \
          ((x)*(x))
```

Präprozessor (3)

- parametrisierte Makros
  - Fallstrick Seiteneffekte

```
#define SQUARE(x) ((x)*(x))
int x = 2;
printf("%d", SQUARE(++x));
```

wird zu

```
printf("%d", ((++x)*(++x));
```

  - ... was wird ausgegeben?
  - Schlimmer noch bei Funktionsaufrufen mit Seiteneffekten!

```
...
printf("%d", SQUARE(launch_missile()));
```

  - Fallunterscheidung mit Ausdrücken

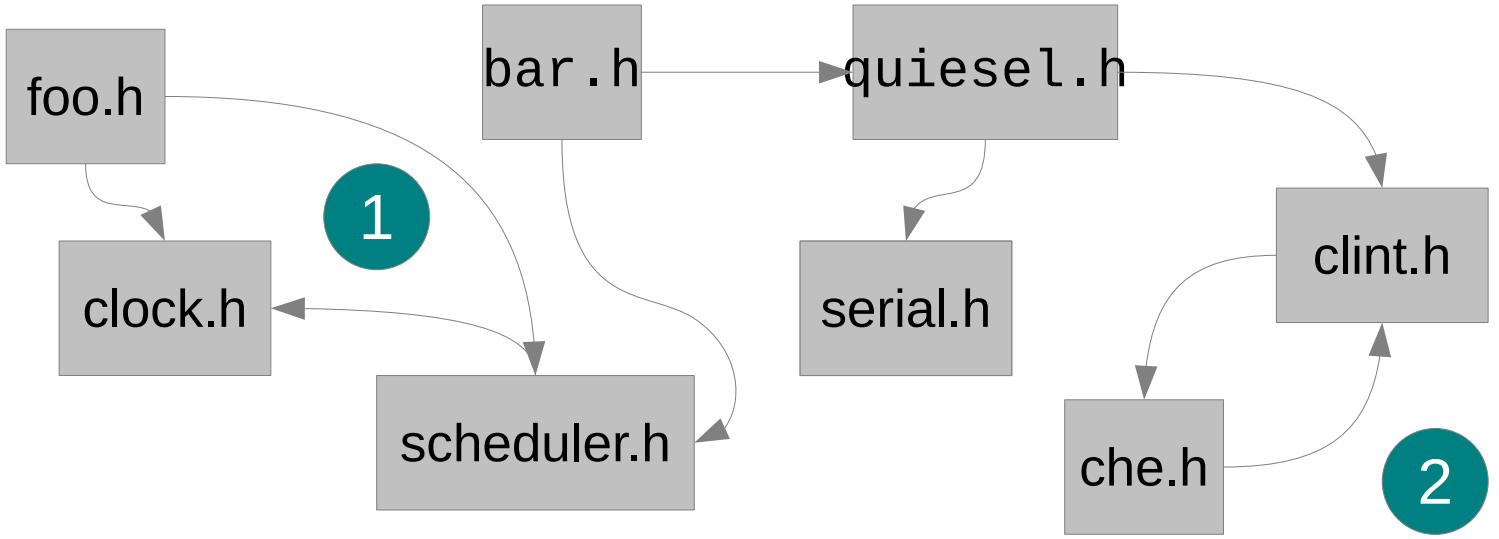
```
#define PFERD 4711
#if PFERD == 815
printf("Pferd.\n");
#else
printf("Kein Pferd.\n");
#endif
```

wird zu

```
printf("Kein Pferd.\n");
```

Include-Wächter

- Szenario: Viele Headerfiles, gegenseitiges Einbinden



  - Inhalte der Headerfiles wiederholen sich (bei 1)
  - Endlosrekursion (bei 2)

- **Lösung:** Include-Wächter →

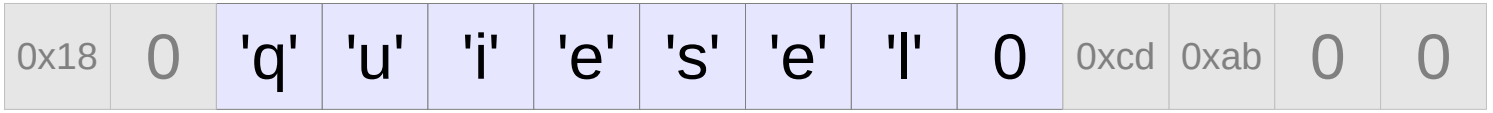
<whatever>.h

```
#ifndef __WHATEVER_H__
#define __WHATEVER_H__
/*inhalt des headers*/
#endif
```

Felder im Speicher

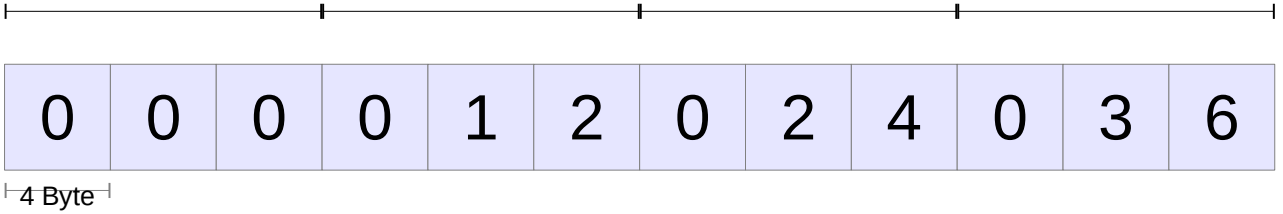
Die einzelnen Elemente der Felder liegen hintereinander im Speicher

```
short int foo = 24;
char wort[] = "quiesel";  /* == {'q','u', ... 'l',0}*/
int kuckuck = 0xABCD;
```



diese können wiederum komplexe Datenstrukturen sein
zum Beispiel ein 3-elementiges Array

```
int mult[4][3] = { {0,0,0},
                   {0,1,2},
                   {0,2,4},
                   {0,3,6} };
```



Felder im Speicher (2)

```c
struct element {
    int x,y;
    char text[16];
};

struct element matrix[3][2] = { { {0,0,"null,null"}, {0,1,"null,eins"} },
                                { {1,0,"eins,null"}, {1,1,"eins,eins"} },
                                { {2,0,"zwei,null"}, {2,1,"zwei,eins"} }
                                };
```

matrix[1][1].x

```
Contents of section .data:
 8049520 00000000 00000000 2c940408 00000000  ....,...........
 8049530 00000000 00000000 00000000 00000000  ................
 8049540 00000000 00000000 6e756c6c 2c6e756c  ........null,nul
 8049550 6c000000 00000000 00000000 01000000  l...............
 8049560 6e756c6c 2c65696e 73000000 00000000  null,eins.......
 8049570 01000000 00000000 65696e73 2c6e756c  ........eins,nul
 8049580 6c000000 00000000 01000000 01000000  l...............
 8049590 65696e73 2c65696e 73000000 00000000  eins,eins.......
 80495a0 02000000 00000000 7a776569 2c6e756c  ........zwei,nul
 80495b0 6c000000 00000000 02000000 01000000  l...............
 80495c0 7a776569 2c65696e 73000000 00000000  zwei,eins.......
```

Arrays und Zeiger

- Array-Bezeichner sind im Prinzip konstante Zeiger auf den Anfang der Felder

```
char text[] = "quiesel";
char *c = text; /* synonym zu &(text[0]) */
*c = 'k'; /* text ist jetzt "kuiesel"! */
```

- Zeiger lassen sich wie Array-Bezeichner verwenden

```
c[1] = 'w';  /* text ist jetzt "kwiesel"! */
```

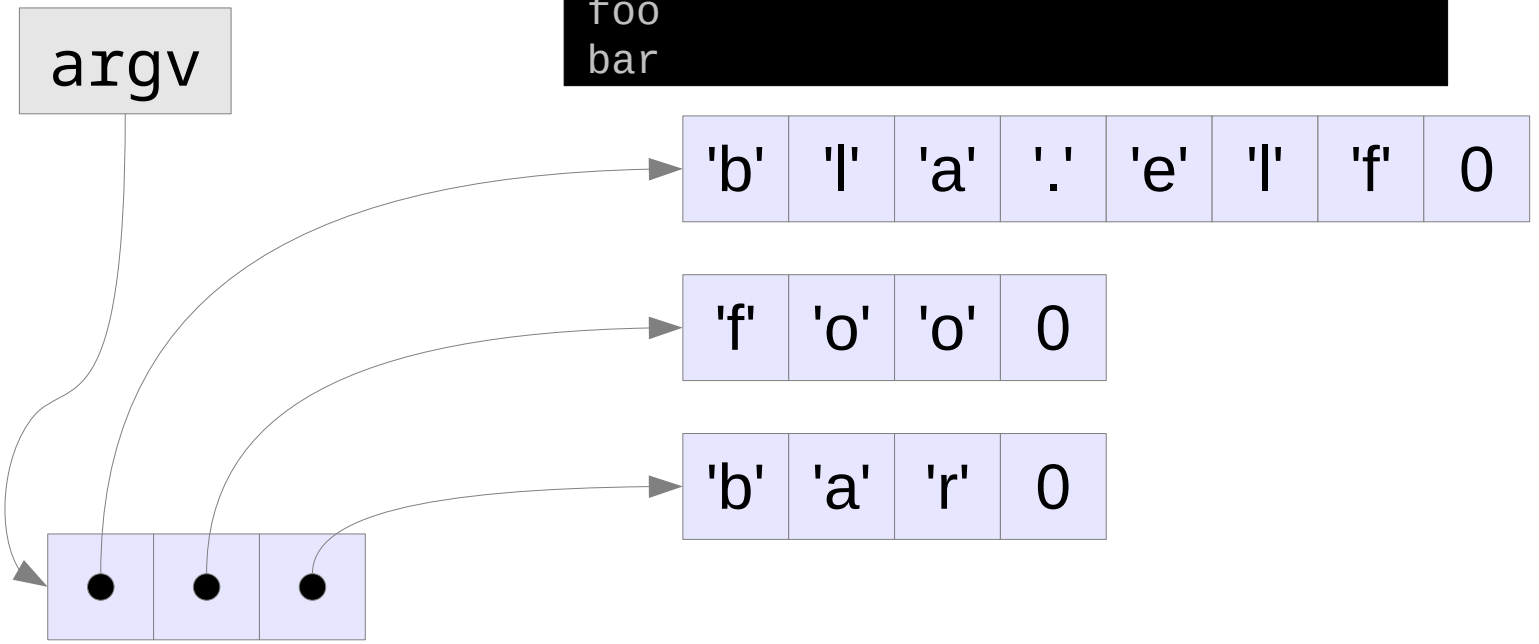
- Umgekehrt gilt dies nicht immer
  - Array-Bezeichner sind keine Variablen sondern Konstanten
  - Sie haben im Gegensatz zu Zeigern keine Adresse im Speicher
  - `&text` ist an sich Blödsinn, liefert aber dieselbe Adresse wie `text`!

```
*text = 'd';  /* text ist jetzt "dwiesel"! */
/* text = c;  WUERDE NICHT KOMPILIEREN! */
c = (char*) &text; /* richtige Adresse, falscher Typ */
```

Kommandozeilenparameter

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

```
js@ios:~/examples$ bla.elf foo bar
bla.elf
foo
bar
```



- `argc` – Anzahl der Parameter (incl. Programmname)
- `argv` – Feld von Zeigern auf die Parameter

Strings

- Literale

```
char foo[] = "quiesel";
int main(void) {
    foo[0] = 'k';
    return 0;
}
```

- läuft durch
- char-Feld mit Inhalt „quiesel“

```
char *foo = "quiesel";
int main(void) {
    foo[0] = 'k';
    return 0;
}
```

```
js@ios:~$ ./literale.elf
Segmentation fault
```

- Zeiger auf read-only-Daten

```
Contents of section .data:
 8049540 71756965 73656c00                quiesel.
```

*

```
Contents of section .rodata:
 8048428 03000000 01000200 71756965  ........quiesel
 73656c00                            .
```

*

- und wie schaut's hiermit aus?

```
int main(void) {
    char foo[] = "quiesel";
    foo[0] = 'k';
    return 0;
}
```

*) zeigt `objdump -s programm.elf`

Strings

- Einlesen von Strings
  - Gefährlich (buffer overflow!)

```
char foo[64];
scanf("%s", foo);
```

  - Besser:

```
char foo[64];
scanf("%63s", foo);
```

- Bibliotheksfunktionen

```
char *strncpy(char *dest, const char *src, size_t sz);
char *strncmp(const char *s1, const char *s2, size_t n);
char *strncat(char *dst, const char *src, size_t sz);
```

  - Arbeiten zeichenweise, bis `'\0'`
  - Ebenfalls potentiell gefährlich
  - Alt: `strcpy, strcmp, strcat`
  - Extra Parameter begrenzt Bearbeitung auf n Zeichen