

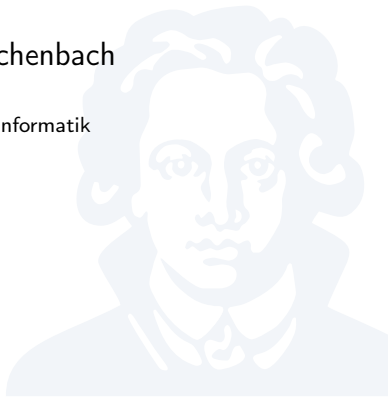
Einführung in die Systemprogrammierung

05

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

20. Mai 2014



Adressen in C

Lade/Schreiboperationen in Maschinensprache erlauben beliebigen Speicherzugriff:

main:

```
la    $t0, main
sw    $zero, $t0(40)  # 10. Befehl überschreiben
lb    $a0, $t0(-1)
li    $v0, 1          # Davorliegendes Byte ausgeben
syscall
```

- Register beinhalten Speicheradressen
- Für Systemprogrammierung nötig:
 - Hardwareansteuerung
 - Maschinencode-Erzeugung

C benötigt beliebigen Zugriff auf beliebige Adressen

Adressen in C

Lade/Schreiboperationen in Maschinensprache erlauben beliebigen Speicherzugriff:

main:

```
la    $t0, main
sw    $zero, $t0(40)  # 10. Befehl überschreiben
lb    $a0, $t0(-1)
li    $v0, 1          # Davorliegendes Byte ausgeben
syscall
```

- Register beinhalten Speicheradressen
- Für Systemprogrammierung nötig:
 - Hardwareansteuerung
 - Maschinencode-Erzeugung

C benötigt beliebigen Zugriff auf beliebige Adressen

C-Typ für Adressen: Zeiger

Zeiger

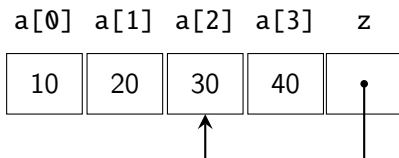
a[0] a[1] a[2] a[3] z

10	20	30	40	
----	----	----	----	--

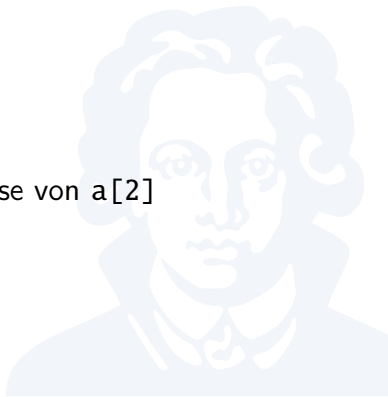
- `int *z`
z hat Typ *Zeiger auf int*



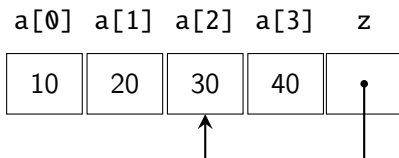
Zeiger



- `int *z`
z hat Typ *Zeiger auf int*
- `z = &a[2]`
Adreßoperator: z nimmt die Adresse von a[2]

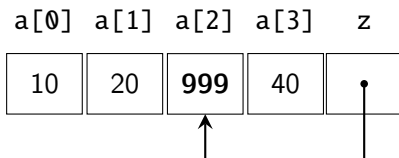


Zeiger



- `int *z`
z hat Typ *Zeiger auf int*
- `z = &a[2]`
Adreßoperator: z nimmt die Adresse von a[2]
- `*z`
Dereferenzierungsoperator: Greift auf `int`-Wert der Adresse von z zu

Zeiger



- `int *z`
z hat Typ *Zeiger auf int*
- `z = &a[2]`
Adreßoperator: z nimmt die Adresse von a[2]
- `*z`
Dereferenzierungsoperator: Greift auf `int`-Wert der Adresse von z zu
`*z = 999`

Operationen auf Zeigern

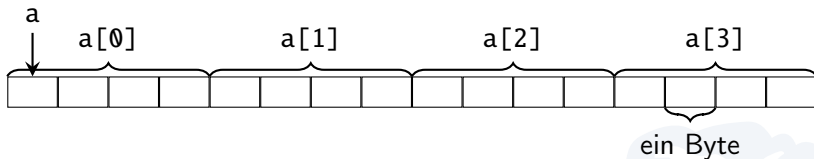
```
int a[4] = {10, 20, 30, 40};
```

```
main() {  
    int *a_zeiger = &a;    // Zeigt auf Anfang  
    int *a_ende = &a[3];  // Zeigt auf Ende  
    while (a_zeiger <= a_ende) {  
        int wert = *a_zeiger; // Speicherzelle lesen  
        a_zeiger += 1;        // Eine int-Speicherzelle weiter  
    }  
}
```

- Typ für „Zeiger auf t “: t^*
- *Adreßoperator* zur Adressierung: $\&lvalue$ (nur lvalues!)
- *Dereferenzierung* zum Auslesen: $*zeiger$

Zeigerarithmetik und `sizeof`

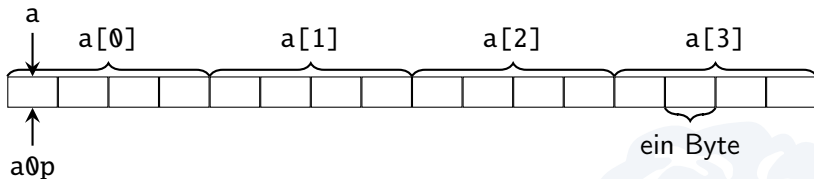
Annahme: `sizeof(int) = 4`



```
int a[4] = ...;
```

Zeigerarithmetik und `sizeof`

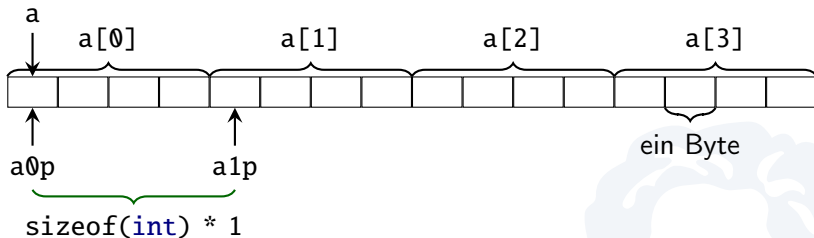
Annahme: `sizeof(int) = 4`



```
int a[4] = ...;  
int *a0p = &a[0];
```

Zeigerarithmetik und sizeof

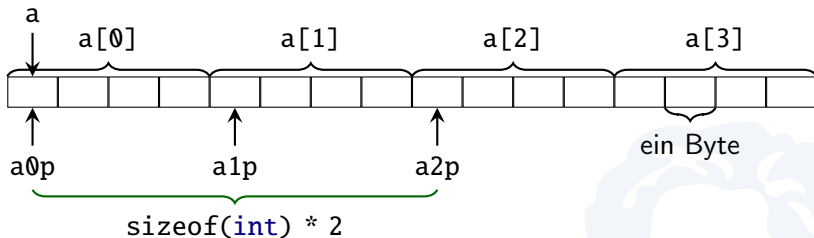
Annahme: `sizeof(int) = 4`



```
int a[4] = ...;  
int *a0p = &a[0];  
int *a1p = a0p + 1;
```

Zeigerarithmetik und sizeof

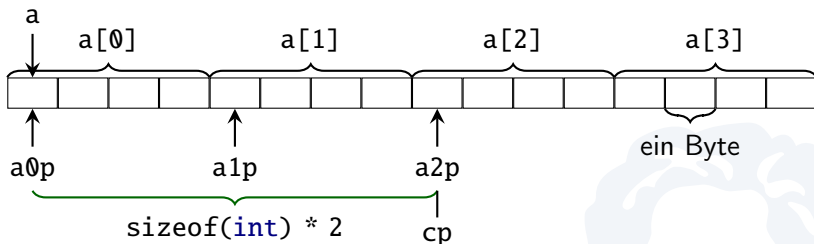
Annahme: `sizeof(int) = 4`



```
int a[4] = ...;  
int *a0p = &a[0];  
int *a1p = a0p + 1;  
int *a2p = a0p + 2;
```

Zeigerarithmetik und sizeof

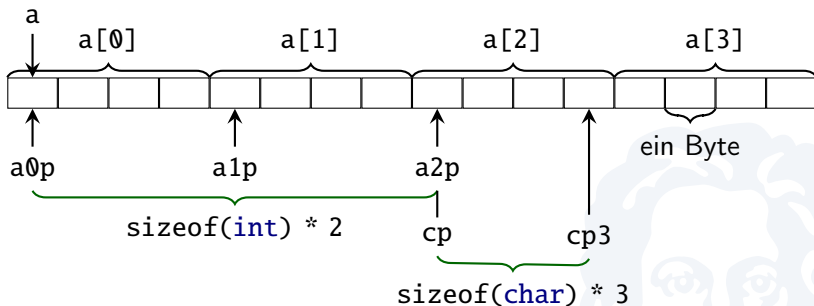
Annahme: `sizeof(int) = 4`, `sizeof(char) = 1`



```
int a[4] = ...;  
int *a0p = &a[0];  
int *a1p = a0p + 1;  
int *a2p = a0p + 2;  
char *cp = (char *)a2p;
```

Zeigerarithmetik und sizeof

Annahme: `sizeof(int) = 4`, `sizeof(char) = 1`



```
int a[4] = ...;
int *a0p = &a[0];
int *a1p = a0p + 1;
int *a2p = a0p + 2;
char *cp = (char *)a2p;
char *cp3 = cp + 3;
```

Zeiger und Arrays

```
int a[4] = ...;
```



Zeiger und Arrays

```
int a[4] = ...;  
int *a0p = a;    // arrays können als Zeiger dienen
```



Zeiger und Arrays

```
int a[4] = ...;
int *a0p = a;    // arrays können als Zeiger dienen
```

		Zeiger (<code>int *v</code>)	Arrays (<code>int v[3]</code>)
Adresse		<i>beliebig</i>	<i>fest zugeordnet</i>
Modifizierbar	<code>v += 1</code>	<i>ja</i>	<i>nein</i>
Initialisierung		<code>int *v = &i</code>	<code>int v[3]={1,2,3}</code>
Lesbar	<code>int i = *v</code>	<i>ja</i>	<i>ja</i>
Schreibbar	<code>*v = 23</code>	<i>ja</i>	<i>ja</i>
Indizierung	<code>v[3]</code>	<i>ja</i>	<i>ja</i>

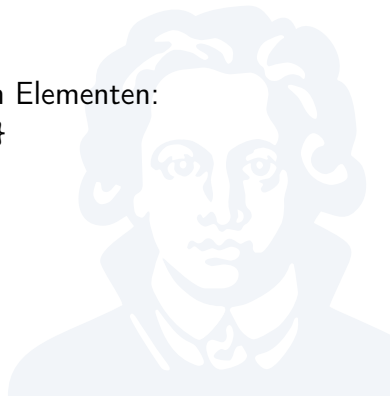
Bequeme Kurzformen:

- Array-Indizierung ist Kurzform für Addition und Dereferenzierung:
 $v[n] == *(v + n) == n[v]$



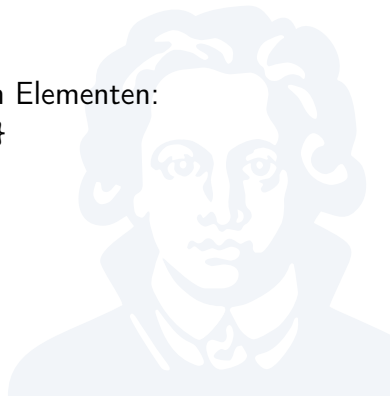
Bequeme Kurzformen:

- Array-Indizierung ist Kurzform für Addition und Dereferenzierung:
 $v[n] == *(v + n) == n[v]$
- Array mit unbekannter Anzahl von Elementen:
`int a[] = { 1, 2, 3, 4, 5 }`



Bequeme Kurzformen:

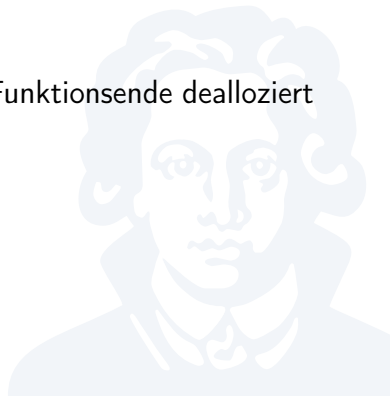
- Array-Indizierung ist Kurzform für Addition und Dereferenzierung:
 $v[n] == *(v + n) == n[v]$
- Array mit unbekannter Anzahl von Elementen:
`int a[] = { 1, 2, 3, 4, 5 }`
- Feldzugriff durch einen Zeiger:
`a->b == (*a).b`



Zeiger und Speicherformen

Zeiger können in *alle* Speicherformen zeigen:

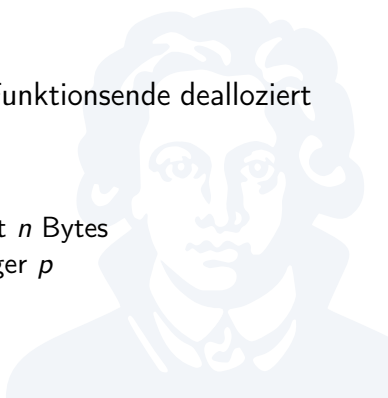
- Statisch:
Zu Programmstart festgelegt
- Stapel:
Bei Funktionsaufruf alloziert, bei Funktionsende dealloziert



Zeiger und Speicherformen

Zeiger können in *alle* Speicherformen zeigen:

- Statisch:
Zu Programmstart festgelegt
- Stapel:
Bei Funktionsaufruf alloziert, bei Funktionsende dealloziert
- Ablagespeicher:
Freie programmatische Kontrolle:
 - `void *malloc(int n)` alloziert n Bytes
 - `free(void *p)` dealloziert Zeiger p



malloc und free

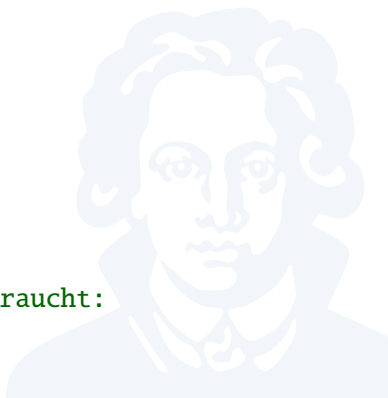
```
// Dynamische Allokierung auf Ablagespeicher
kundenkonto_t *konto =
    malloc(sizeof(kundenkonto_t));
// Platz für ein Konto

...

// Zugriff auf Feld im Konto:
konto->kundennr = 1000;

...

// Konten werden nicht mehr gebraucht:
free(konto);
```



malloc und free

```
// Dynamische Allokierung auf Ablagespeicher
kundenkonto_t *konto =
    malloc(sizeof(kundenkonto_t));
// Platz für ein Konto

...

// Zugriff auf Feld im Konto:
konto->kundennr = 1000;

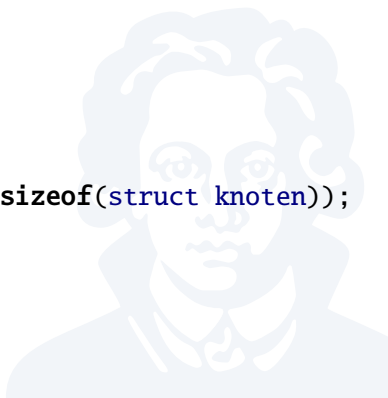
...

// Konten werden nicht mehr gebraucht:
free(konto);
```


Der NULL-Zeiger

```
// Struktur für Binärbaum
struct knoten {
    // '*' muß bei mehreren Deklarationen wiederholt werden:
    struct knoten *links, *rechts;
    int wert;
};

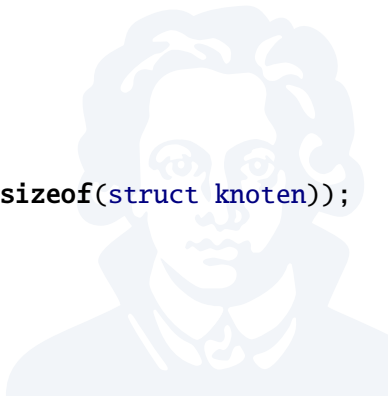
main() {
    struct knoten *wurzel = malloc(sizeof(struct knoten));
    wurzel->wert = 0;
    wurzel->links =
    wurzel->rechts =
```



Der NULL-Zeiger

```
#include<stddef.h>
// Struktur für Binärbaum
struct knoten {
    // '*' muß bei mehreren Deklarationen wiederholt werden:
    struct knoten *links, *rechts;
    int wert;
};

main() {
    struct knoten *wurzel = malloc(sizeof(struct knoten));
    wurzel->wert = 0;
    wurzel->links = NULL;
    wurzel->rechts = NULL;
}
```



Der NULL-Zeiger

```
#include<stddef.h>
// Struktur für Binärbaum
struct knoten {
    // '*' muß bei mehreren Deklarationen wiederholt werden:
    struct knoten *links, *rechts;
    int wert;
};

main() {
    struct knoten *wurzel = malloc(sizeof(struct knoten));
    wurzel->wert = 0;
    wurzel->links = NULL;
    wurzel->rechts = NULL;
```

NULL (aus `stddef.h`) ist designierter „Zeiger nach Nirgendwo“

Zeiger auf Funktionen

- C unterstützt Zeiger auf alles, was im Speicher liegt
- Der Maschinencode von Funktionen liegt im Speicher
- Typsystem unterstützt *Funktionszeiger*

```
int f(unsigned, char);
```

```
main() {  
  
    p = &f;  
  
}
```



Zeiger auf Funktionen

- C unterstützt Zeiger auf alles, was im Speicher liegt
- Der Maschinencode von Funktionen liegt im Speicher
- Typsystem unterstützt *Funktionszeiger*

```
int f(unsigned, char);
```

```
main() {  
    int (*p)(unsigned, char);  
    p = &f;  
}
```

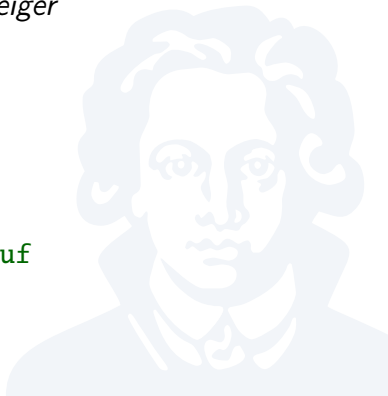


Zeiger auf Funktionen

- C unterstützt Zeiger auf alles, was im Speicher liegt
- Der Maschinencode von Funktionen liegt im Speicher
- Typsystem unterstützt *Funktionszeiger*

```
int f(unsigned, char);
```

```
main() {  
    int (*p)(unsigned, char);  
    p = &f;  
    (*p)(0, 'x'); // Funktionsaufruf  
}
```



Zusammenfassung: Zeiger

- Zeiger können auf beliebige Speicheradressen zeigen
- Adresse laden per Adreßoperator: `&lvalue`
- Speicherzugriff per Dereferenzierungsoperator: `*zeiger`
- Arrays können fast alles, was Zeiger können
- Zeigerarithmetik:
 - Zeiger verschieben mit `+` und `-`
 - Zeiger auf `t` ändert Adresse immer um Vielfaches von `sizeof(t)`
- **NULL**: designierter Zeiger, der auf kein Objekt zeigt
- Kombinierte Dereferenzierung/Strukturzugriff:
`struktur->feld`
- Zeiger auf Funktionen:
 - Deklaration:
`rückgabetyt (*zeiger)(parametertypen);`
 - Aufruf: `(*zeiger)(tatsächliche parameter)`

Kontext: Zeiger in anderen Sprachen

- Zeiger in C (und C++) sind sehr flexibel:
 - Beliebiger Speicherzugriff
 - Keine Einschränkungen durch Laufzeitsystem
- Diese Flexibilität ist für Systemprogrammierung nötig
- Andere Sprachen (Java, Haskell, ...) haben andere Anforderungen:
 - *Typsicherheit*: Alle Typfehler werden vom Übersetzer oder Laufzeitsystem entdeckt
 - *Automatische Speicherverwaltung*: Ablagespeicher nicht per `free` freigegeben (häufige Fehlerquelle!) sondern automatisch
- Zeiger machen Typsicherheit und automatische Speicherverwaltung unmöglich

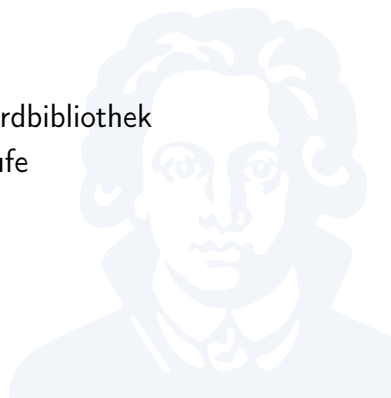
Alle Operatoren

Operatoren (stärker bindende oben)

	Arg.	Assoz.
(Ausdruck) [...] -> .	2	links
! ~ ++ - + - (typ) * & sizeof	1	rechts
* / %	2	links
+ -	2	links
<< >>	2	links
< <= >= >	2	links
== !=	2	links
&	2	links
^	2	links
	2	links
&&	2	links
	2	links
... ? ... : ...	3	rechts
= += -= *= /= %= <<= >>= = &= ^=	2	rechts
,	2	links

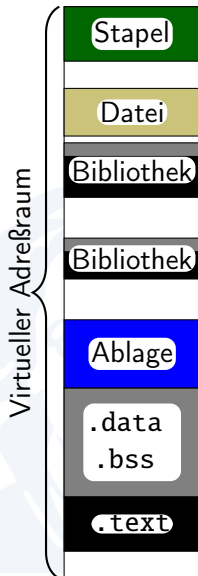
Das Laufzeitsystem der Sprache C

- Speicheraufteilung (Wiederholung)
- Speicherlokationen von Variablen
- Speicher und Seitentabelle
- Speicheroperationen der C-Standardbibliothek
- Stapelspeicher und Funktionsaufrufe
- Ablagespeicher



Wiederholung: Speicher zur Laufzeit

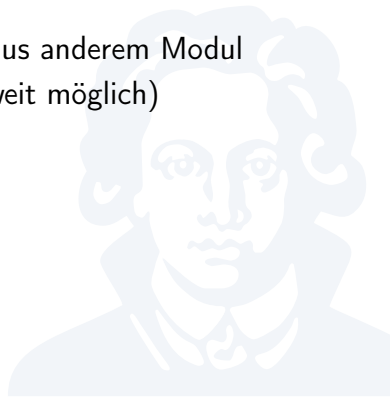
- Stapelspeicher
 - Lokale Variablen
 - Parameter
 - Rücksprungadressen
- Ablagespeicher
 - malloc etc.
- `.data`, `.bss`:
 - Statische/Globale Variablen
- `.text`:
 - Funktionen



Variablen im Speicher

Speicherklassen: geben u.a. an, wo Variablen im Speicher liegen

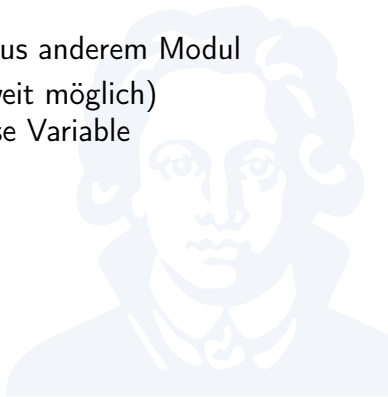
- **auto:** Auf dem Stapelspeicher
- **static:** Im statischen Speicher
- **extern:** Im statischen Speicher, aus anderem Modul
- **register:** In einem Register (soweit möglich)



Variablen im Speicher

Speicherklassen: geben u.a. an, wo Variablen im Speicher liegen

- **auto**: Auf dem Stapelspeicher
- **static**: Im statischen Speicher
- **extern**: Im statischen Speicher, aus anderem Modul
- **register**: In einem Register (soweit möglich)
Verbietet den **&**-Operator auf diese Variable



Variablen im Speicher

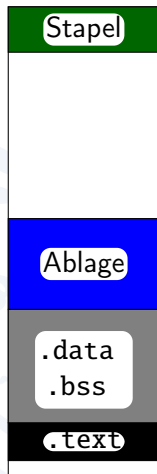
Speicherklassen: geben u.a. an, wo Variablen im Speicher liegen

- **auto:** Auf dem Stapelspeicher
- **static:** Im statischen Speicher
- **extern:** Im statischen Speicher, aus anderem Modul
- **register:** In einem Register (soweit möglich)
Verbietet den **&**-Operator auf diese Variable
- Ohne Angabe?
 - In Funktionskörper: wie **auto**
 - Außerhalb des Funktionskörpers: ähnlich **static**
(Unterschied: Sichtbar für andere Module)

Speicher und Variablen

```
void f(int x)
{
    int y;
    auto int z;
    int *a =
        malloc(sizeof(int));
    register int b;
    static int c;
}
int x;
static int d;
```

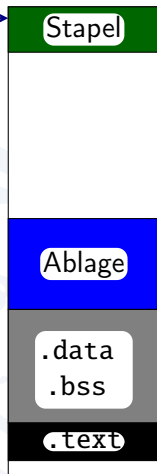
CPU



Speicher und Variablen

```
void f(int x)
{
    int y;
    auto int z;
    int *a =
        malloc(sizeof(int));
    register int b;
    static int c;
}
int x;
static int d;
```

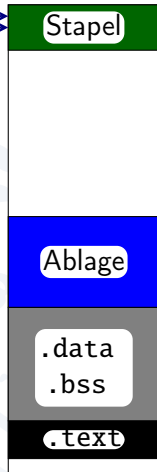
CPU



Speicher und Variablen

```
void f(int x)
{
    int y;
    auto int z;
    int *a =
        malloc(sizeof(int));
    register int b;
    static int c;
}
int x;
static int d;
```

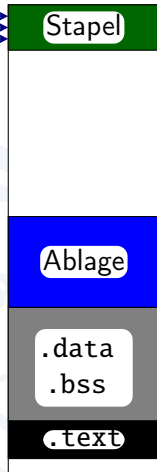
CPU



Speicher und Variablen

```
void f(int x)
{
    int y;
    auto int z;
    int *a =
        malloc(sizeof(int));
    register int b;
    static int c;
}
int x;
static int d;
```

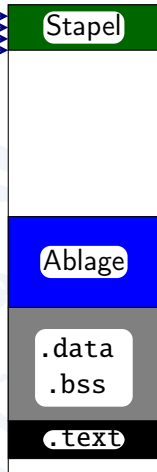
CPU



Speicher und Variablen

```
void f(int x)
{
    int y;
    auto int z;
    int *a = malloc(sizeof(int));
    register int b;
    static int c;
}
int x;
static int d;
```

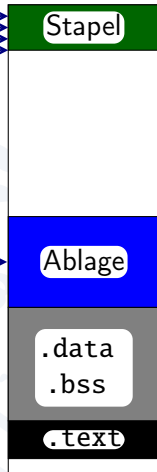
CPU



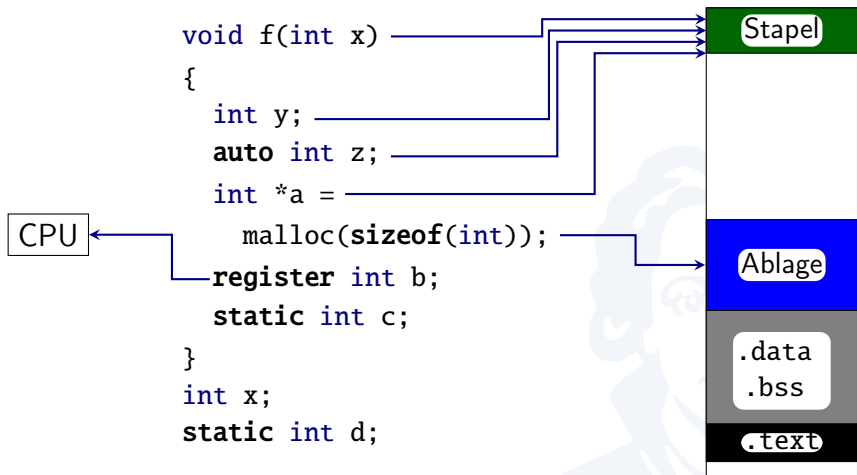
Speicher und Variablen

```
void f(int x)
{
    int y;
    auto int z;
    int *a =
        malloc(sizeof(int));
    register int b;
    static int c;
}
int x;
static int d;
```

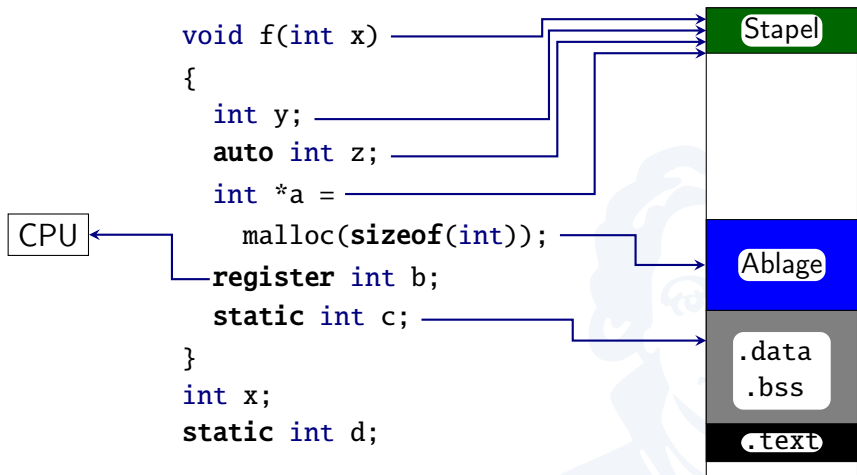
CPU



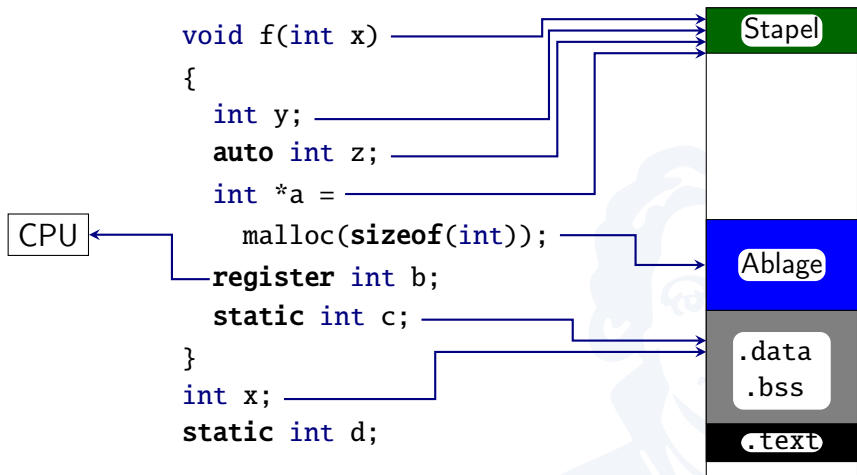
Speicher und Variablen



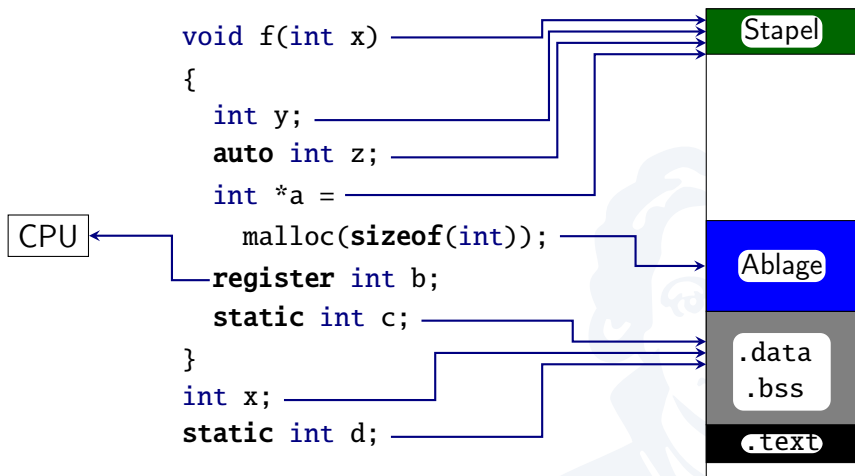
Speicher und Variablen



Speicher und Variablen

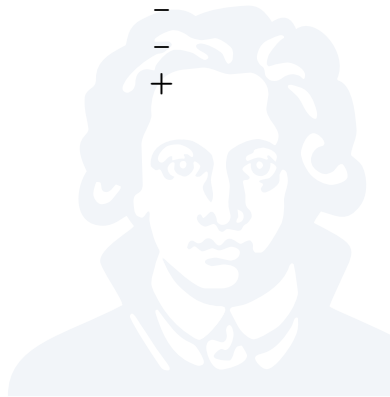


Speicher und Variablen



Speicherklassen für verschiedene Variablen

	Toplevel	In Funktionskörper	Parameter
auto	-	+	-
static	+	+	-
extern	+	+	-
register	-	+	+

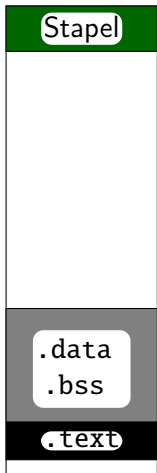


Speicherklassen für verschiedene Variablen

	Toplevel	In Funktionskörper	Parameter
auto	-	+	-
static	+	+	-
extern	+	+	-
register	-	+	+

Inhalte von uninitialisierten Variablen sind nur für **static** vorbestimmt, sonst implementierungsabhängig (!)

Speicher, Seitentabelle, Systemaufrufe in UNIX



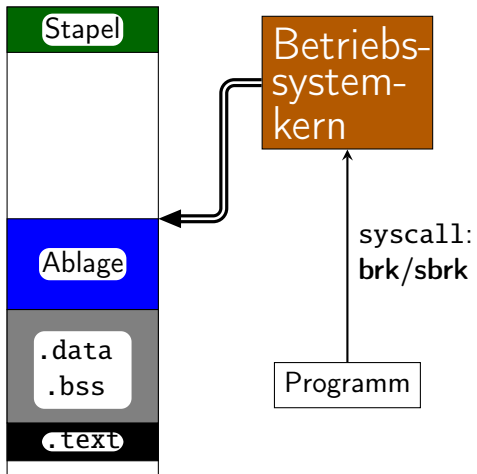
Betriebs-
system-
kern

Programm

- Programmstart: Stapel, statischer Speicher, Programmspeicher

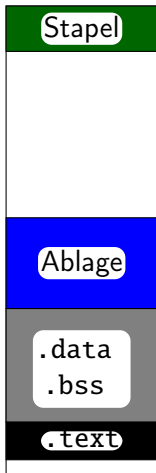


Speicher, Seitentabelle, Systemaufrufe in UNIX



- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert

Speicher, Seitentabelle, Systemaufrufe in UNIX

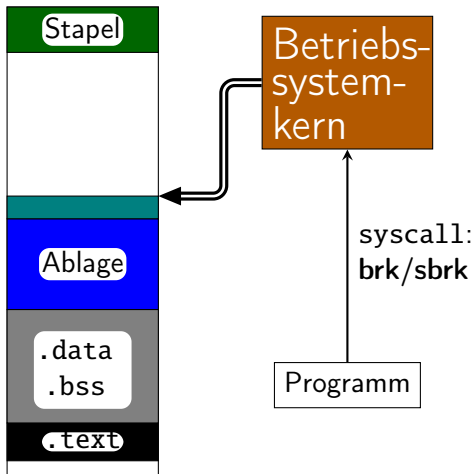


Betriebs-
system-
kern

Programm

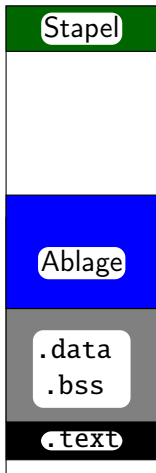
- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert

Speicher, Seitentabelle, Systemaufrufe in UNIX



- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert/wächst

Speicher, Seitentabelle, Systemaufrufe in UNIX

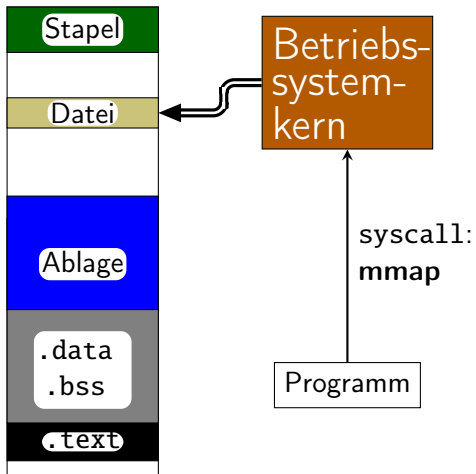


Betriebs-
system-
kern

Programm

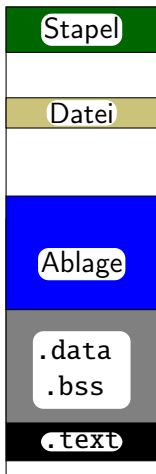
- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert/wächst

Speicher, Seitentabelle, Systemaufrufe in UNIX



- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert/wächst
- Systemaufruf `mmap`: Datei oder leerer Speicher wird in Speicher abgebildet

Speicher, Seitentabelle, Systemaufrufe in UNIX

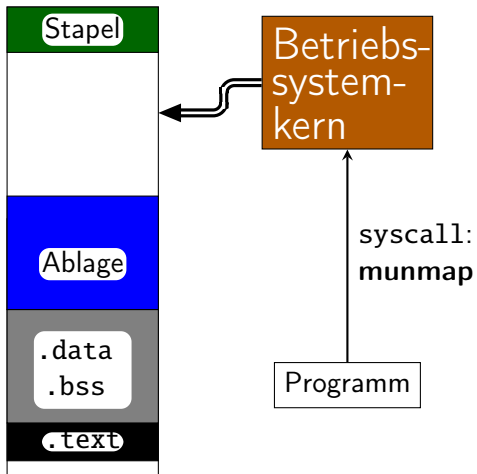


Betriebssystem-kern

Programm

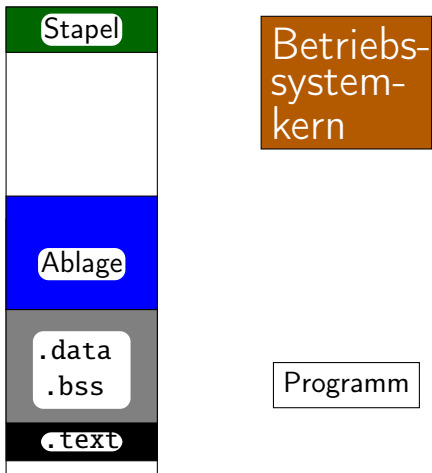
- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert/wächst
- Systemaufruf `mmap`: Datei oder leerer Speicher wird in Speicher abgebildet

Speicher, Seitentabelle, Systemaufrufe in UNIX



- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert/wächst
- Systemaufruf `mmap`: Datei oder leerer Speicher wird in Speicher abgebildet
- Systemaufruf `munmap`: Abbildung wird entfernt

Speicher, Seitentabelle, Systemaufrufe in UNIX



- Programmstart: Stapel, statischer Speicher, Programmspeicher
- Systemaufruf `brk/sbrk`: Ablage alloziert/wächst
- Systemaufruf `mmap`: Datei oder leerer Speicher wird in Speicher abgebildet
- Systemaufruf `munmap`: Abbildung wird entfernt

Diese syscalls werden meist nur implizit von der Standardbibliothek ausgeführt

Zusammenfassung: Speicher in C

- Stapelspeicher (lesen, schreiben)
 - Funktionsparameter, lokale Variablen, Rücksprungadresse
- Ablagespeicher (lesen, schreiben)
 - `malloc()`, `free()`
- Programmspeicher (`.text`) (ausführen, lesen)
- Statischer Speicher (`.data`, `.bss`) (lesen, schreiben)
- Zahlreiche syscalls zur Manipulation von Seitentabelle
 - `brk/sbrk`, `mmap/munmap/mprotect`

Ablagespeicheroperationen

```
#include<stdlib.h>
```

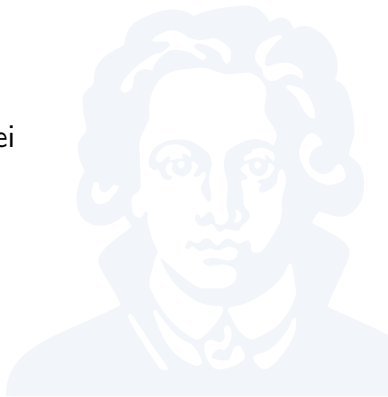
- Systemaufrufe `brk`, `sbrk` allozieren neue Seiten für Ablagespeicher
(Normalerweise nicht selbst aufrufen)



Ablagespeicheroperationen

```
#include<stdlib.h>
```

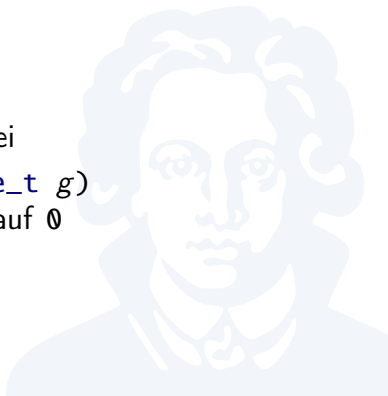
- Systemaufrufe `brk`, `sbrk` allozieren neue Seiten für Ablagespeicher
(Normalerweise nicht selbst aufrufen)
- `void *malloc(size_t g)`
Alloziert g Bytes
- `free(void *p)`
Gibt allozierten Zeiger p wieder frei



Ablagespeicheroperationen

```
#include<stdlib.h>
```

- Systemaufrufe `brk`, `sbrk` allozieren neue Seiten für Ablagespeicher
(Normalerweise nicht selbst aufrufen)
- `void *malloc(size_t g)`
Alloziert g Bytes
- `free(void *p)`
Gibt allozierten Zeiger p wieder frei
- `void *calloc(size_t n, size_t g)`
Alloziert $n \times g$ Bytes, initialisiert auf `0`



Ablagespeicheroperationen

```
#include<stdlib.h>
```

- Systemaufrufe `brk`, `sbrk` allozieren neue Seiten für Ablagespeicher
(Normalerweise nicht selbst aufrufen)
- `void *malloc(size_t g)`
Alloziert g Bytes
- `free(void *p)`
Gibt allozierten Zeiger p wieder frei
- `void *calloc(size_t n, size_t g)`
Alloziert $n \times g$ Bytes, initialisiert auf 0
- `void *realloc(void *p, size_t g')`
Allozierter Zeiger p (alte Größe g) erhält neue Größe g' :
 - Rückgabewert kann neuer oder alter Zeiger sein
 - $\min(g, g')$ Bytes werden kopiert, falls nötig

Blockspeicheroperationen

`#include<string.h>`

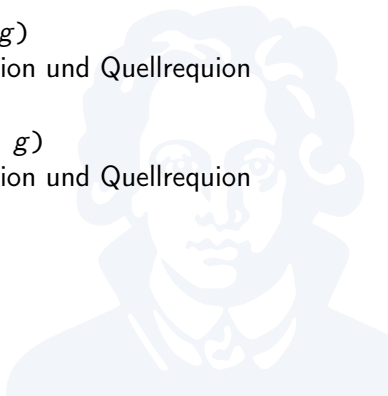
- Verschieben und Kopieren von Speicherblöcken ist manchmal in der Systemprogrammierung nötig
- Effiziente Implementierungen nichttrivial, daher in C-Standardbibliothek



Blockspeicheroperationen

`#include<string.h>`

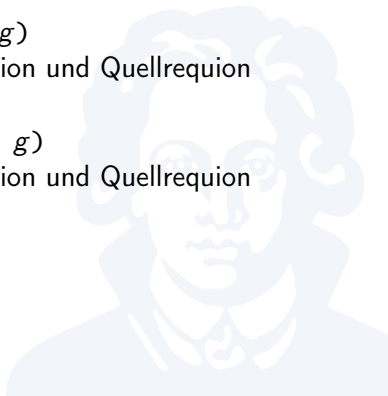
- Verschieben und Kopieren von Speicherblöcken ist manchmal in der Systemprogrammierung nötig
- Effiziente Implementierungen nichttrivial, daher in C-Standardbibliothek
- `memcpy(void *z, void *q, int g)`
Kopiert g Bytes ab q nach z . Zielregion und Quellregion dürfen nicht überlappen.
- `memmove(void *z, void *q, int g)`
Kopiert g Bytes ab q nach z . Zielregion und Quellregion können überlappen.



Blockspeicheroperationen

`#include<string.h>`

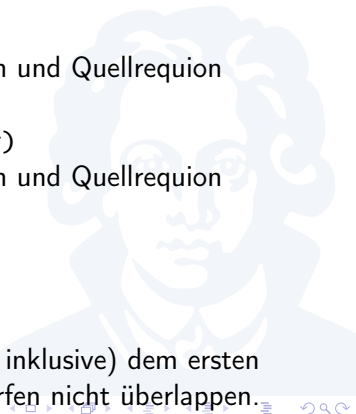
- Verschieben und Kopieren von Speicherblöcken ist manchmal in der Systemprogrammierung nötig
- Effiziente Implementierungen nichttrivial, daher in C-Standardbibliothek
- `memcpy(void *z, void *q, int g)`
Kopiert g Bytes ab q nach z . Zielregion und Quellregion dürfen nicht überlappen.
- `memmove(void *z, void *q, int g)`
Kopiert g Bytes ab q nach z . Zielregion und Quellregion können überlappen.
- `memset(void *z, int n, int g)`
Setzt g Bytes ab z auf Byte-Wert n .



Blockspeicheroperationen

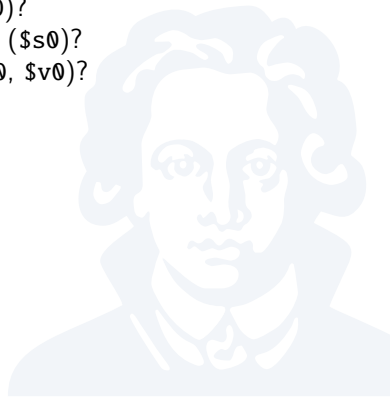
`#include<string.h>`

- Verschieben und Kopieren von Speicherblöcken ist manchmal in der Systemprogrammierung nötig
- Effiziente Implementierungen nichttrivial, daher in C-Standardbibliothek
- `memcpy(void *z, void *q, int g)`
Kopiert g Bytes ab q nach z . Zielregion und Quellregion dürfen nicht überlappen.
- `memmove(void *z, void *q, int g)`
Kopiert g Bytes ab q nach z . Zielregion und Quellregion können überlappen.
- `memset(void *z, int n, int g)`
Setzt g Bytes ab z auf Byte-Wert n .
- `strcpy(void *z, void *q)`
Kopiert Speicher ab q nach z , bis (und inklusive) dem ersten `0`-Byte. Zielregion und Quellregion dürfen nicht überlappen.



Funktionsaufrufe (Wiederholung)

- Funktionsaufrufbehandlung ist komplex:
 - Unterscheidung bei Registern:
 - Vom Aufrufer zu sichern ($\$t0$)?
 - Vom Aufgerufenen zu sichern ($\$s0$)?
 - Zur Übergabe verwendet ($\$a0, \$v0$)?
 - Aufrufstapelverwaltung



Funktionsaufrufe (Wiederholung)

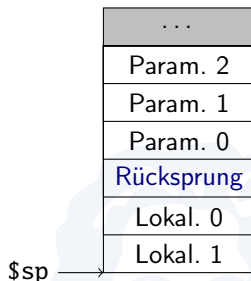
- Funktionsaufrufbehandlung ist komplex:
 - Unterscheidung bei Registern:
 - Vom Aufrufer zu sichern ($\$t0$)?
 - Vom Aufgerufenen zu sichern ($\$s0$)?
 - Zur Übergabe verwendet ($\$a0, \$v0$)?
 - Aufrufstapelverwaltung

Betrachten wir vereinfachend Aufrufe ohne Register

Funktionsaufrufe in C

(Annahme: Aufrufkonventionen ohne Register)

- Stapel sichert Aufrufdaten („Aufrufstapel“):
 - Parameter
 - Lokale (nicht-statische/globale) Variablen
 - Rücksprungadresse (bei Bedarf)
- Zugriff auf Stapelinhalte: $\$sp + c$
 - c ist Konstante, da Stapelgröße bekannt

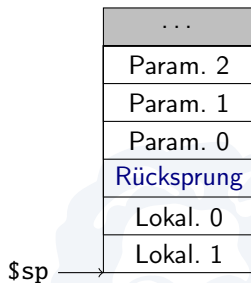


Funktionsaufrufe in C

(Annahme: Aufrufkonventionen ohne Register)

- Stapel sichert Aufrufdaten („Aufrufstapel“):
 - Parameter
 - Lokale (nicht-statische/globale) Variablen
 - Rücksprungadresse (bei Bedarf)
- Zugriff auf Stapelinhalte: $\$sp + c$
 - c ist Konstante, da Stapelgröße bekannt

```
int f(int x) {  
    int y;  
    int a[x*x];  
    ...  
}
```

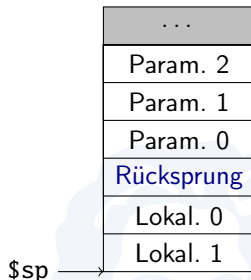


Funktionsaufrufe in C

(Annahme: Aufrufkonventionen ohne Register)

- Stapel sichert Aufrufdaten („Aufrufstapel“):
 - Parameter
 - Lokale (nicht-statische/globale) Variablen
 - Rücksprungadresse (bei Bedarf)
- Zugriff auf Stapelinhalte: $\$sp + c$
 - c ist Konstante, da Stapelgröße bekannt

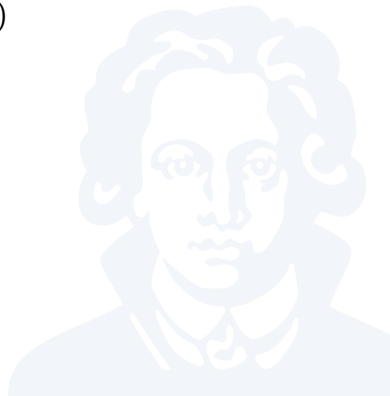
```
int f(int x) {  
    int y;  
    int a[x*x];  
...  
}
```



Arrays mit dynamischer Größe \Rightarrow Stapelgröße unbekannt!

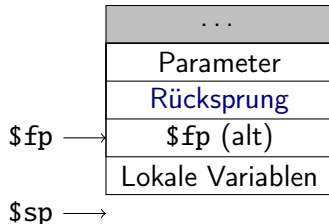
Der Rahmenzeiger `$fp`

- `$sp + c`-Zugriff nur möglich mit *bekannter* Stapelgröße
- Einige Dinge bewirken *unbekannte* Stapelgröße:
 - Arrays mit dynamischer Größe (`int a[x*x];`)
 - Stapel-inlining (Optimierung)
 - `obstack` (gcc-Spracherweiterung)



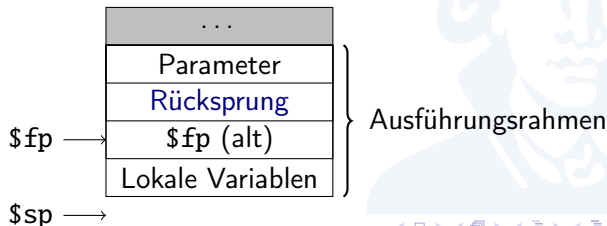
Der Rahmenzeiger `$fp`

- `$sp + c`-Zugriff nur möglich mit *bekannter* Stapelgröße
- Einige Dinge bewirken *unbekannte* Stapelgröße:
 - Arrays mit dynamischer Größe (`int a[x*x];`)
 - Stapel-inlining (Optimierung)
 - `obstack` (gcc-Spracherweiterung)
- Lösung: *Rahmenzeigerregister* `$fp` bleibt konstant
 - Zugriff auf Parameter/lokale Variablen per `$fp + c`



Der Ausführungsrahmen

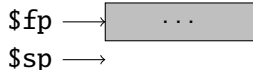
- *Ausführungsrahmen*:
(*activation record*, stack frame)
 - Parameter
 - Gesicherte Rücksprungadresse
 - Gesicherter alter Rahmenzeiger
 - Lokale Variablen
- Reihenfolge von Rücksprungadresse, Rahmenzeiger, Lokalen Variablen ist beliebig (kein Einfluß auf Kommunikation mit anderen Subroutinen)



Funktionsaufrufe und Stapel

```
int main()
...
    f(2); //main+1c
...
int f(int x) {
    int y = x * 2;
    return g(y+1, 7); //f+8
}
int g(int a, int b) {
    int c = a * b;
    if (c < 0) {
        int d = b - a;
        return d * (c - d);
    }
    return c;
}
```

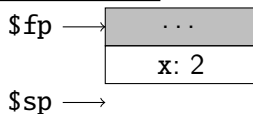
Aufrufstapel



Funktionsaufrufe und Stapel

```
int main()
...
    f(2); //main+1c
...
int f(int x) {
    int y = x * 2;
    return g(y+1, 7); //f+8
}
int g(int a, int b) {
    int c = a * b;
    if (c < 0) {
        int d = b - a;
        return d * (c - d);
    }
    return c;
}
```

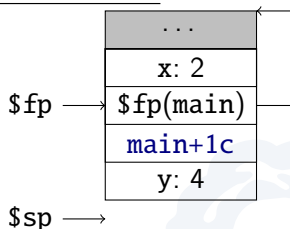
Aufrufstapel



Funktionsaufrufe und Stapel

```
int main()
...
    f(2); //main+1c
...
int f(int x) {
    int y = x * 2;
    return g(y+1, 7); //f+8
}
int g(int a, int b) {
    int c = a * b;
    if (c < 0) {
        int d = b - a;
        return d * (c - d);
    }
    return c;
}
```

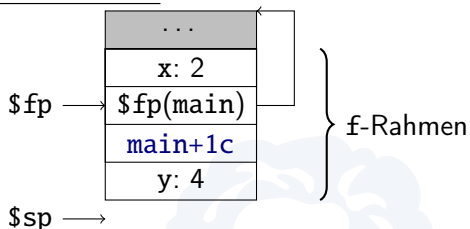
Aufrufstapel



Funktionsaufrufe und Stapel

```
int main()
...
    f(2); //main+1c
...
int f(int x) {
    int y = x * 2;
    return g(y+1, 7); //f+8
}
int g(int a, int b) {
    int c = a * b;
    if (c < 0) {
        int d = b - a;
        return d * (c - d);
    }
    return c;
}
```

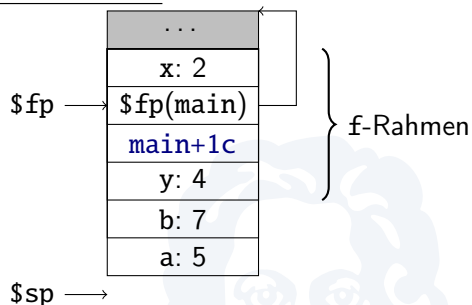
Aufrufstapel



Funktionsaufrufe und Stapel

```
int main()
...
    f(2); //main+1c
...
int f(int x) {
    int y = x * 2;
    return g(y+1, 7); //f+8
}
int g(int a, int b) {
    int c = a * b;
    if (c < 0) {
        int d = b - a;
        return d * (c - d);
    }
    return c;
}
```

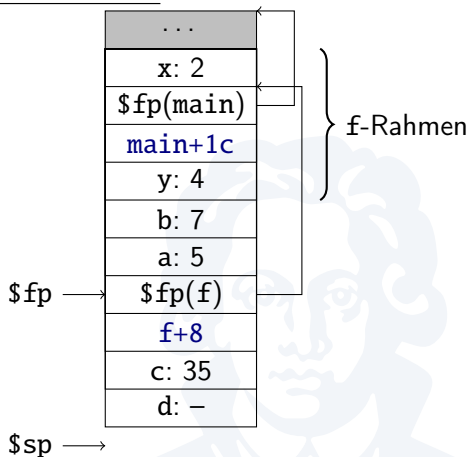
Aufrufstapel



Funktionsaufrufe und Stapel

```
int main()
...
    f(2); //main+1c
...
int f(int x) {
    int y = x * 2;
    return g(y+1, 7); //f+8
}
int g(int a, int b) {
    int c = a * b;
    if (c < 0) {
        int d = b - a;
        return d * (c - d);
    }
    return c;
}
```

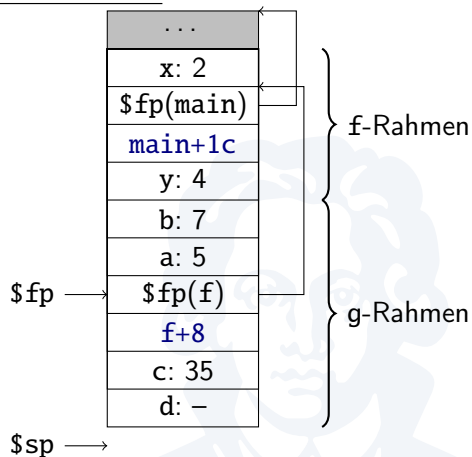
Aufrufstapel



Funktionsaufrufe und Stapel

```
int main()
...
    f(2); //main+1c
...
int f(int x) {
    int y = x * 2;
    return g(y+1, 7); //f+8
}
int g(int a, int b) {
    int c = a * b;
    if (c < 0) {
        int d = b - a;
        return d * (c - d);
    }
    return c;
}
```

Aufrufstapel



Aufruf (ohne Param.-Register)

Aufrufer:

- Sichert vom Aufrufer zu sichernde Register ($\$t0, \dots$)
- Vergrößert $\$sp$ für Parameter
- Schreibt Parameter
- Springt (jal)

Rücksprung



Aufruf (ohne Param.-Register)

Aufrufer:

- Sichert vom Aufrufer zu sichernde Register ($\$t0, \dots$)
- Vergrößert $\$sp$ für Parameter
- Schreibt Parameter
- Springt (jal)

Aufgerufener:

- Sichert:
 - $\$fp$ des Aufrufers
 - Rücksprungadresse
 - vom Aufgerufenen zu sichernde Register ($\$s0, \dots$)
- Setzt $\$fp = \sp
- Vergrößert $\$sp$ für Ausführungsrahmen

Rücksprung



Aufruf (ohne Param.-Register)

Aufrufer:

- Sichert vom Aufrufer zu sichernde Register ($\$t0, \dots$)
- Vergrößert $\$sp$ für Parameter
- Schreibt Parameter
- Springt (jal)

Aufgerufener:

- Sichert:
 - $\$fp$ des Aufrufers
 - Rücksprungsadresse
 - vom Aufgerufenen zu sichernde Register ($\$s0, \dots$)
- Setzt $\$fp = \sp
- Vergrößert $\$sp$ für Ausführungsrahmen

Rücksprung

Aufgerufener:

- Lade Rückgabewert in Rückgaberegister ($\$v0$)
- Stellt $\$sp$ wieder her ($\$sp = \fp)
- Läd aus Ausführungsrahmen:
 - Rahmenzeiger $\$fp$ des Aufrufers
 - Rücksprungsadresse
 - vom Aufgerufenen gesicherte Register ($\$s0, \dots$)

Aufrufer:

- Verringert $\$sp$ um Parameter

Funktionsaufrufe: Sonderfälle

- Zuletzt betrachtet: Allgemeinfall
- Spezialfälle:
 - *Feste Stapelgröße:*
Kann $\$sp+c$ statt $\$fp+c$ verwenden, $\$fp$ für anderes
 - *Alles paßt in Register, keine Rekursion:*
Kann auf Ausführungsrahmen verzichten
 - *Rückgabewert zu groß für Rückgaberegister:*
Aufrufer reserviert Stapelplatz bei Aufruf, Rückgabewert auf Stapel geschrieben
 - *Inlining-Optimierung:*
Übersetzer 'kopiert' aufgerufene Funktion in Aufruferkörper, kein Aufruf nötig

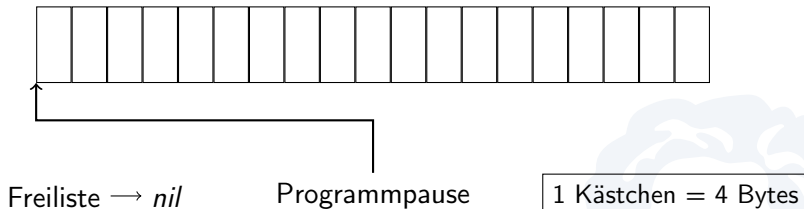
Programmpause und Ablagespeicher

- *Programmpause* markiert oberes Ende des Ablagespeichers
- Bedarf erhöht: Systemaufrufe `brk`, `sbrk`
- Abstand zu Adresse der ersten Bibliothek idealer größer als verfügbarer Speicher
 - Einfach bei 64 Bit
 - 32 Bit: festgesetzte Maximalgröße
- Programmpause meist indirekt über Systembibliothek verwaltet (`malloc()/free()`)

Programmpause



Der First Fit-Algorithmus für Ablagespeicher



Der First Fit-Algorithmus für Ablagespeicher



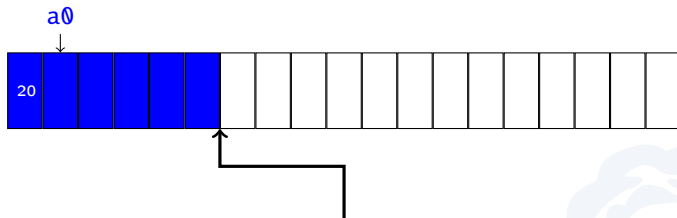
Freiliste → *nil*

Programmpause

1 Kästchen = 4 Bytes

```
a0 = malloc(20); (4 Bytes pro Block)
```

Der First Fit-Algorithmus für Ablagespeicher



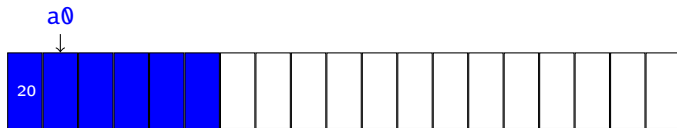
Freiliste \rightarrow *nil*

Programmpause

1 Kästchen = 4 Bytes

```
a0 = malloc(20); (4 Bytes pro Block)
```

Der First Fit-Algorithmus für Ablagespeicher



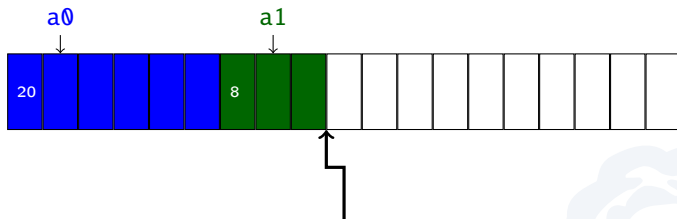
Freiliste \rightarrow *nil*

Programmpause

1 Kästchen = 4 Bytes

```
a0 = malloc(20); (4 Bytes pro Block)  
a1 = malloc(8);
```

Der First Fit-Algorithmus für Ablagespeicher



Freiliste \rightarrow *nil*

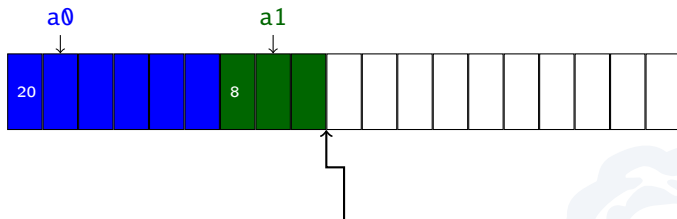
Programmpause

1 Kästchen = 4 Bytes

```
a0 = malloc(20); (4 Bytes pro Block)
```

```
a1 = malloc(8);
```

Der First Fit-Algorithmus für Ablagespeicher



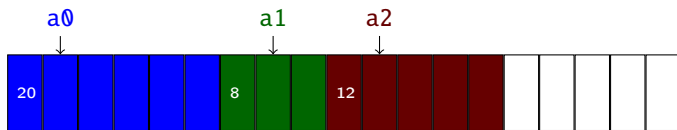
Freiliste \rightarrow *nil*

Programmpause

1 Kästchen = 4 Bytes

```
a0 = malloc(20); (4 Bytes pro Block)  
a1 = malloc(8);  
a2 = malloc(12);
```

Der First Fit-Algorithmus für Ablagespeicher



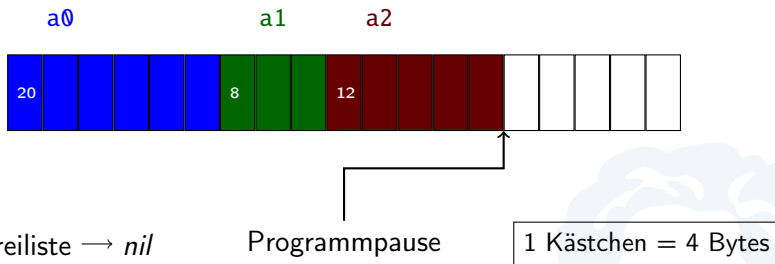
Freiliste \rightarrow *nil*

Programmpause

1 Kästchen = 4 Bytes

```
a0 = malloc(20); (4 Bytes pro Block)
a1 = malloc(8);
a2 = malloc(12);
```

Der First Fit-Algorithmus für Ablagespeicher



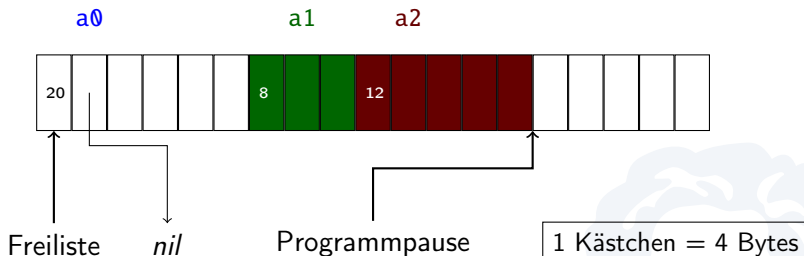
Freiliste → *nil*

Programmpause

1 Kästchen = 4 Bytes

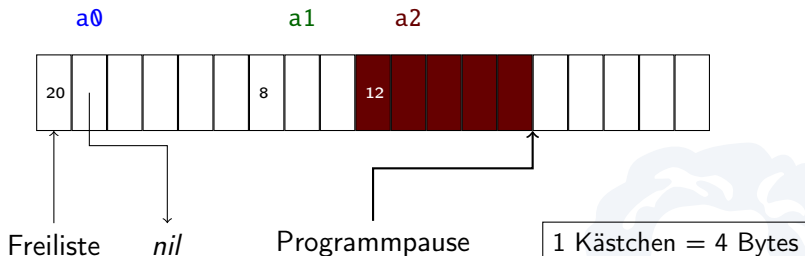
```
a0 = malloc(20); (4 Bytes pro Block)
a1 = malloc(8);
a2 = malloc(12);
free(a0);
```


Der First Fit-Algorithmus für Ablagespeicher



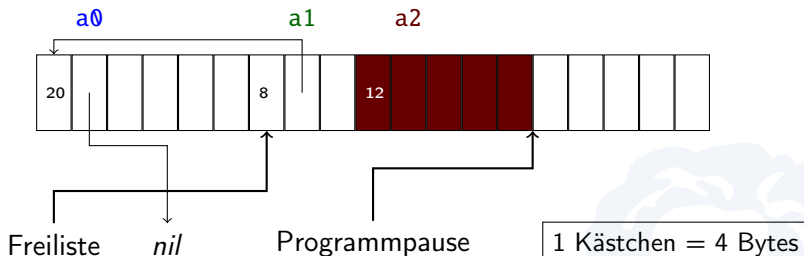
```
a0 = malloc(20); (4 Bytes pro Block)
a1 = malloc(8);
a2 = malloc(12);
free(a0);
```

Der First Fit-Algorithmus für Ablagespeicher



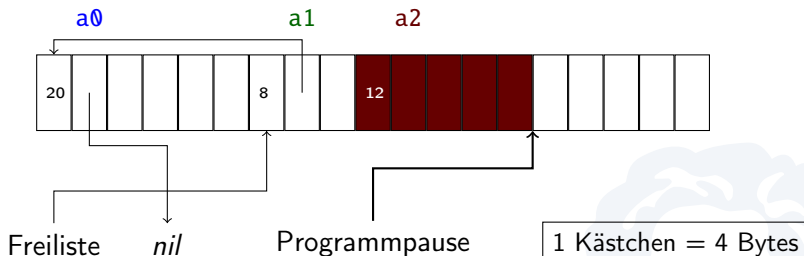
```
a0 = malloc(20); (4 Bytes pro Block)
a1 = malloc(8);
a2 = malloc(12);
free(a0);
free(a1);
```

Der First Fit-Algorithmus für Ablagespeicher



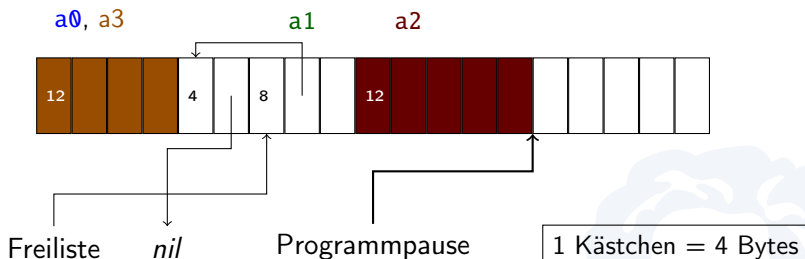
```
a0 = malloc(20); (4 Bytes pro Block)  
a1 = malloc(8);  
a2 = malloc(12);  
free(a0);  
free(a1);
```

Der First Fit-Algorithmus für Ablagespeicher



```
a0 = malloc(20); (4 Bytes pro Block)
a1 = malloc(8);
a2 = malloc(12);
free(a0);
free(a1);
a3 = malloc(12);
```

Der First Fit-Algorithmus für Ablagespeicher



```

a0 = malloc(20); (4 Bytes pro Block)
a1 = malloc(8);
a2 = malloc(12);
free(a0);
free(a1);
a3 = malloc(12);

```

Probleme des First Fit-Algorithmus

- Allokierungszeit $O(k)$ mit k Einträgen in Freiliste
- Speicherfragmentierung über Zeit:
 - Wiederverwendete Blöcke können nur kleiner werden
 - (zu) kleine, weniger nützliche Blöcke sammeln sich an Ende der Freiliste an
 - Verhältnis von Verwaltungsdaten zu genutztem Speicher nimmt tendentiell zu
 - Sprünge bei Blocksuche „verseuchen“ Caches
- „ideale“ Speicherverwaltung weiterhin offenes Forschungsthema

Vergleich: Speicherformen

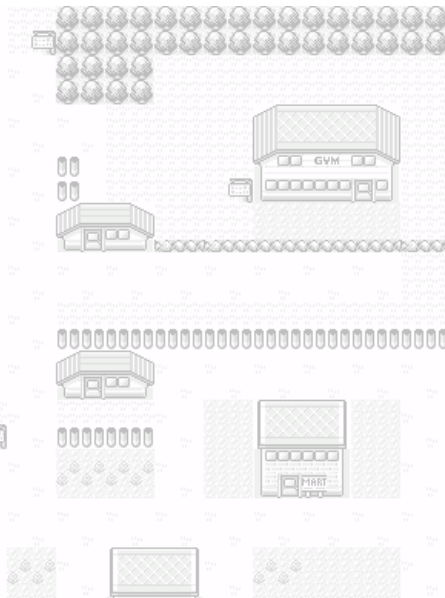
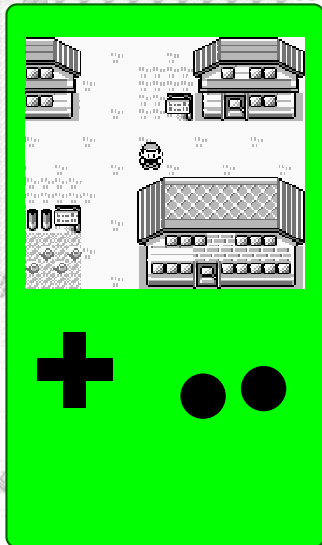
Eigenschaft	Ablage	Stapel	Statisch
Allozierung	malloc	Funktionsaufruf	Variablendeklaration
Deallozierung	free	Rücksprung	—
Größenfestlegung	Laufzeit	Laufzeit	Übersetzungszeit

Rückblick: Laufzeitumgebung

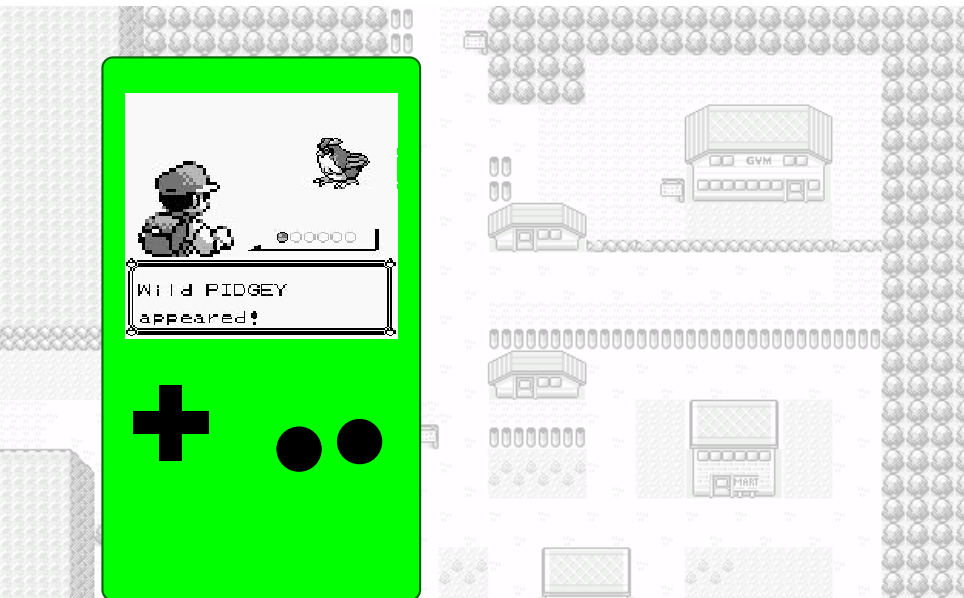
Laufzeitumgebung besteht aus:

- Nichtprivilegiert (*user space*):
 - .text, .data, .bss
 - Stapelspeicher
 - Ablagespeicher
 - Obige beinhalten:
 - Bibliotheksdienste (z.B. Speicherverwaltung)
 - Programmiersprachenspezifische Dienste
 - Registerinhalte
- Betriebssystemkern (*kernel space*) mit Prozeßkontext:
 - Seitentabelle
 - Offene Dateien
 - Offene Netzwerkverbindungen
 - Kommunikation mit anderen Prozessen
 - Sicherungskopien der Register, falls nötig

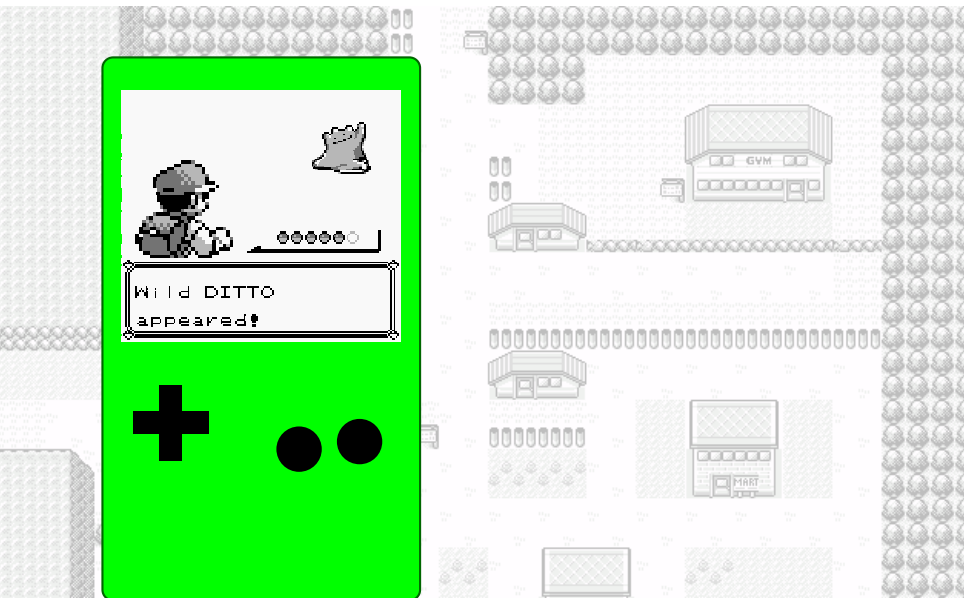
Speicher in Eingebetteten Systemen



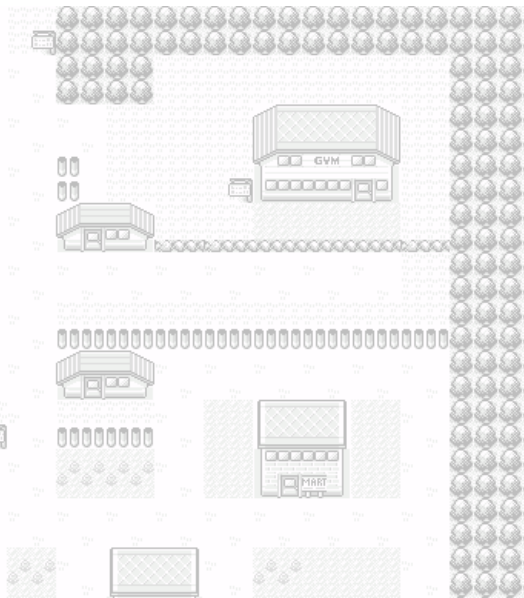
Speicher in Eingebetteten Systemen



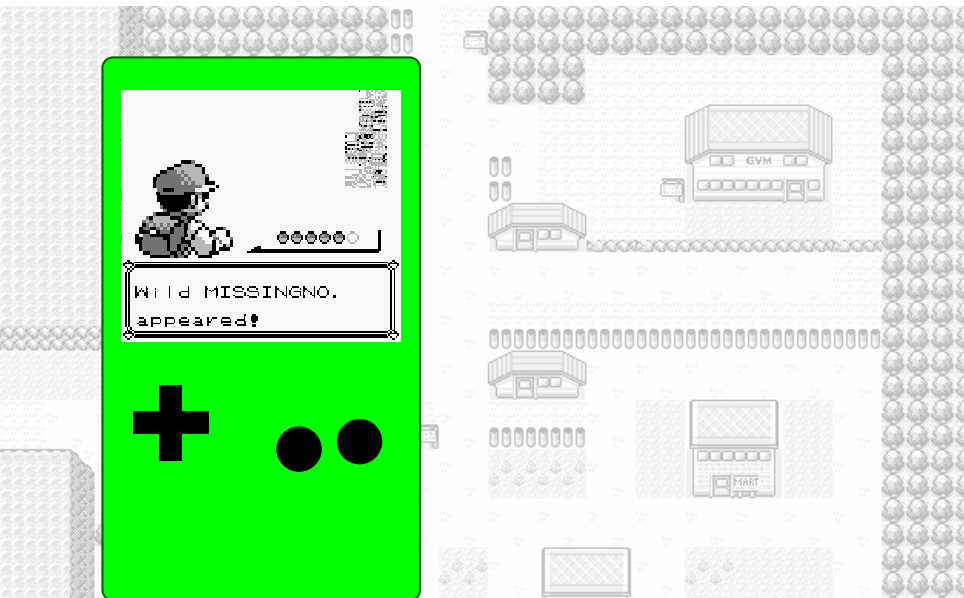
Speicher in Eingebetteten Systemen



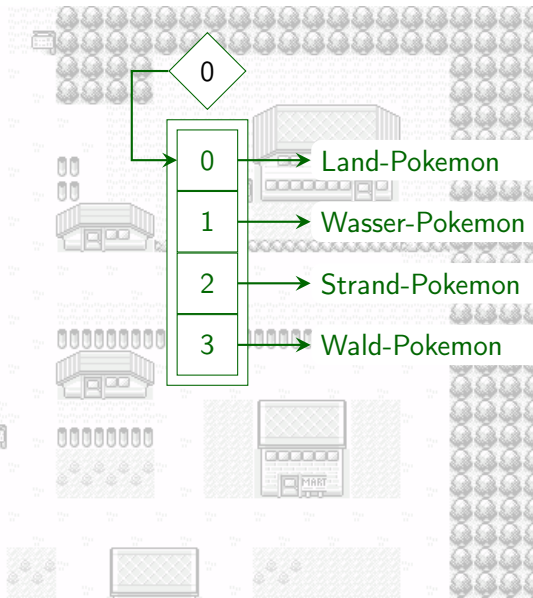
Speicher in Eingebetteten Systemen



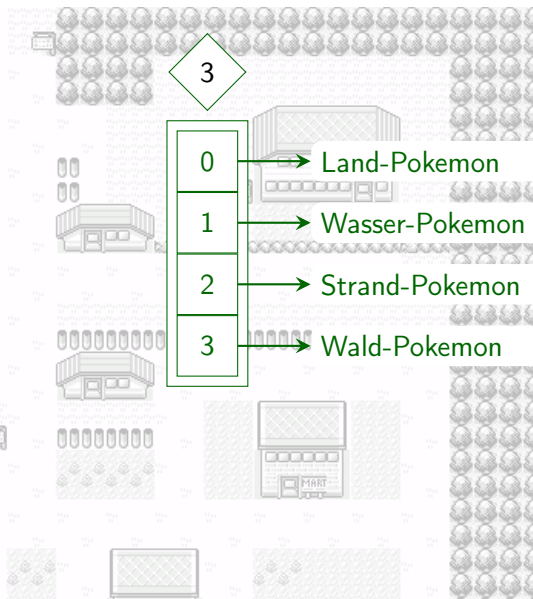
Speicher in Eingebetteten Systemen



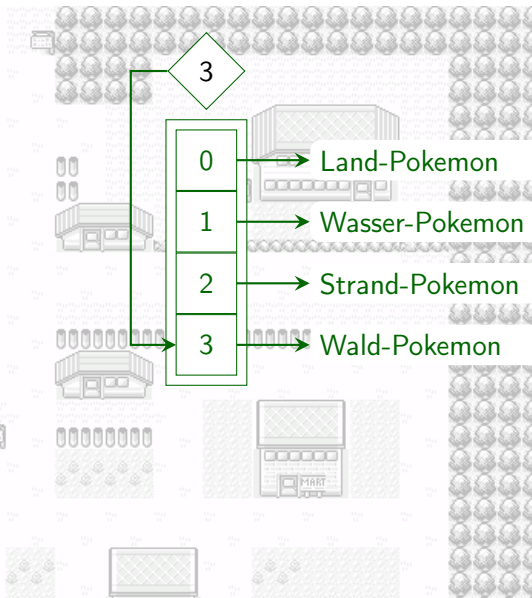
Speicher in Eingebetteten Systemen



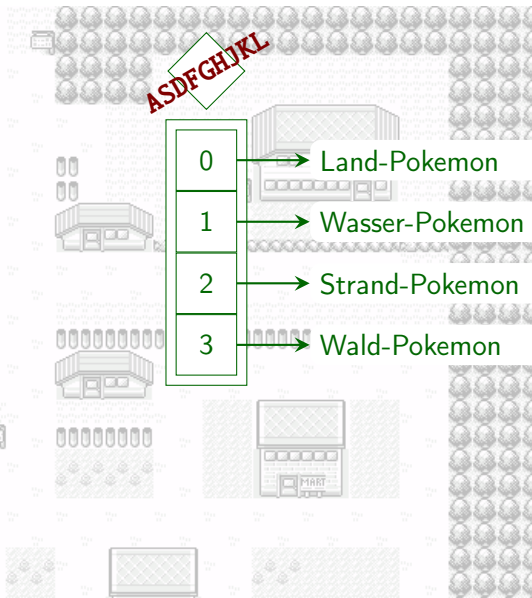
Speicher in Eingebetteten Systemen



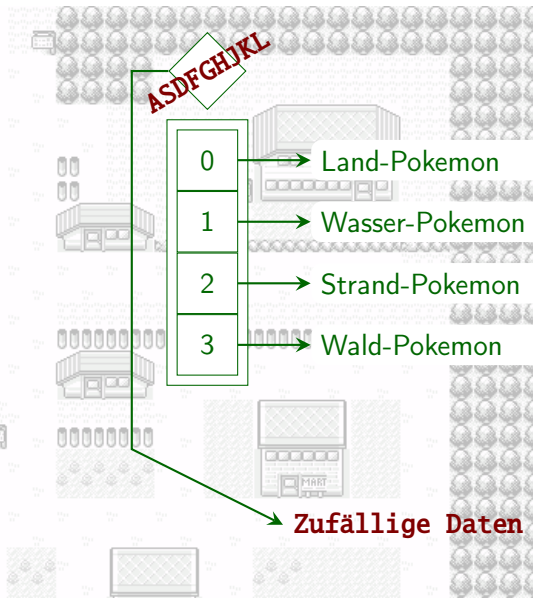
Speicher in Eingebetteten Systemen



Speicher in Eingebetteten Systemen



Speicher in Eingebetteten Systemen



Speicherfehler

- Fehler in Zeigerarithmetik
- Lese-/Schreibzugriff auf deallozierten Speicher
 - „Hängender Zeiger“ (dangling pointer): zeigt unbeabsichtigt auf deallozierten Speicherbereich
- Besonders viele Fehlermöglichkeiten mit Ablagespeicher:
 - free vergessen
 - free auf schon deallozierten Block
 - free vor Speicherzugriff

Speicherfehler führen zu Mißbrauch des Speichers

Speicherfehler: Folgen

Speicherfehler können folgende Konsequenzen haben:

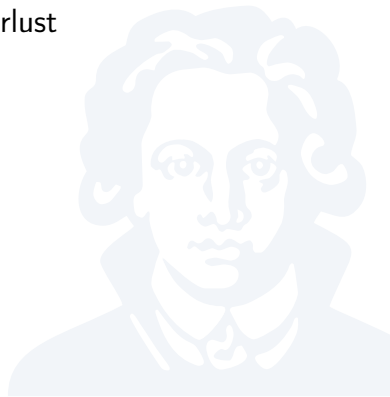
- Segmentierungsfehler
Betriebssystem entdeckt Speicherfehler



Speicherfehler: Folgen

Speicherfehler können folgende Konsequenzen haben:

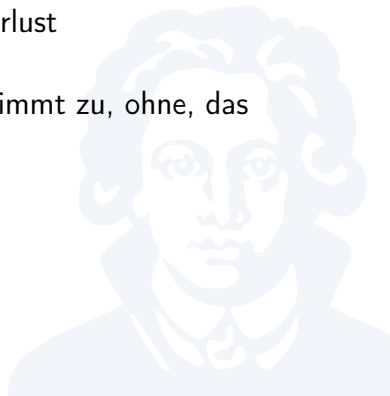
- Segmentierungsfehler
Betriebssystem entdeckt Speicherfehler
- Nicht ausgerichteter Speicherzugriff
Busfehler oder Geschwindigkeitsverlust



Speicherfehler: Folgen

Speicherfehler können folgende Konsequenzen haben:

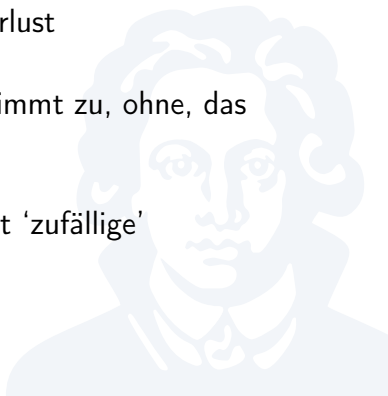
- Segmentierungsfehler
Betriebssystem entdeckt Speicherfehler
- Nicht ausgerichteter Speicherzugriff
Busfehler oder Geschwindigkeitsverlust
- Speicherleck
Speicherbedarf des Programmes nimmt zu, ohne, das Speicher wirklich verwendet wird



Speicherfehler: Folgen

Speicherfehler können folgende Konsequenzen haben:

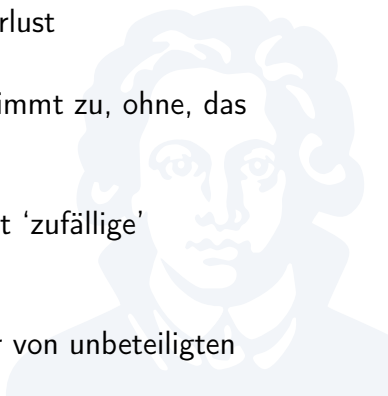
- Segmentierungsfehler
Betriebssystem entdeckt Speicherfehler
- Nicht ausgerichteter Speicherzugriff
Busfehler oder Geschwindigkeitsverlust
- Speicherleck
Speicherbedarf des Programmes nimmt zu, ohne, das Speicher wirklich verwendet wird
- Ungültiger Wert
Programmcode liest unbeabsichtigt 'zufällige' Speicherinhalte



Speicherfehler: Folgen

Speicherfehler können folgende Konsequenzen haben:

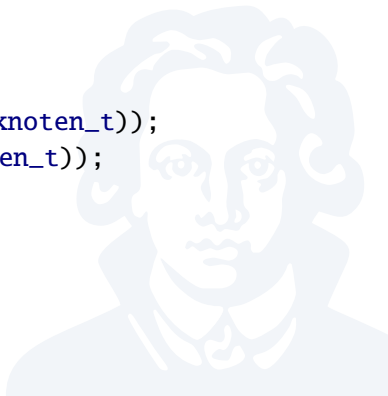
- Segmentierungsfehler
Betriebssystem entdeckt Speicherfehler
- Nicht ausgerichteter Speicherzugriff
Busfehler oder Geschwindigkeitsverlust
- Speicherleck
Speicherbedarf des Programmes nimmt zu, ohne, das Speicher wirklich verwendet wird
- Ungültiger Wert
Programmcode liest unbeabsichtigt 'zufällige' Speicherinhalte
- Speicherkorruption
Programmcode verändert Speicher von unbeteiligten Modulen



Speicherlecks

```
typedef struct k {  
    int wert;  
    struct k *vor;  
    struct k *nach;  
}
```

```
⇒knoten_t *a = calloc(1,sizeof(knoten_t));  
a->nach = calloc(1,sizeof(knoten_t));  
free(a);
```



Speicherlecks

```
typedef struct k {  
    int wert;  
    struct k *vor;  
    struct k *nach;  
}
```

wert: 0
vor: NULL
nach: NULL

```
knoten_t *a = calloc(1, sizeof(knoten_t));  
⇒ a->nach = calloc(1, sizeof(knoten_t));  
free(a);
```

Speicherlecks

```
typedef struct k {  
    int wert;  
    struct k *vor;  
    struct k *nach;  
}
```

wert: 0
vor: NULL
nach:

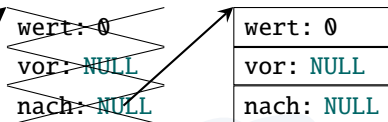
wert: 0
vor: NULL
nach: NULL

```
knoten_t *a = calloc(1, sizeof(knoten_t));  
a->nach = calloc(1, sizeof(knoten_t));  
⇒ free(a);
```

Speicherlecks

```
typedef struct k {  
    int wert;  
    struct k *vor;  
    struct k *nach;  
}
```

```
knoten_t *a = calloc(1, sizeof(knoten_t));  
a->nach = calloc(1, sizeof(knoten_t));  
free(a);
```



Speicherlecks

```
typedef struct k {  
    int wert;  
    struct k *vor;  
    struct k *nach;  
}
```

```
knoten_t *a = calloc(1, sizeof(knoten_t));  
a->nach = calloc(1, sizeof(knoten_t));  
free(a);
```

~~wert: 0~~

~~vor: NULL~~

~~nach: NULL~~

wert: 0

vor: NULL

nach: NULL

Speicherleck: Speicher kann nicht dealloziert werden, ist permanent unbrauchbar

Speicherkorruption

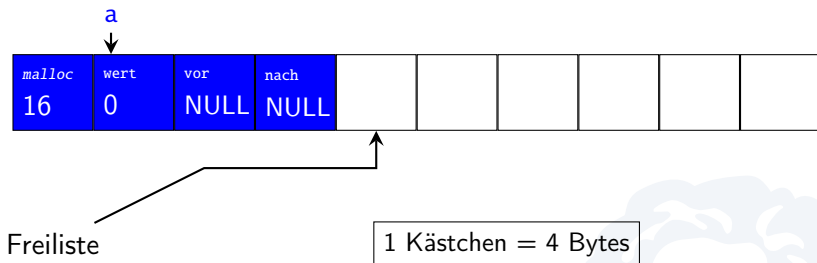


Freiliste

1 Kästchen = 4 Bytes

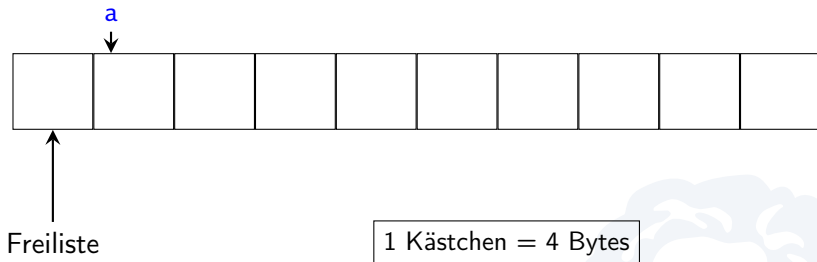
```
⇒ knoten_t *a = calloc(1, sizeof(knoten_t));  
   free(a);  
   double *d = calloc(1, sizeof(double));  
   int *i = calloc(1, sizeof(int));  
   a->nach = calloc(1, sizeof(knoten_t));
```

Speicherkorruption



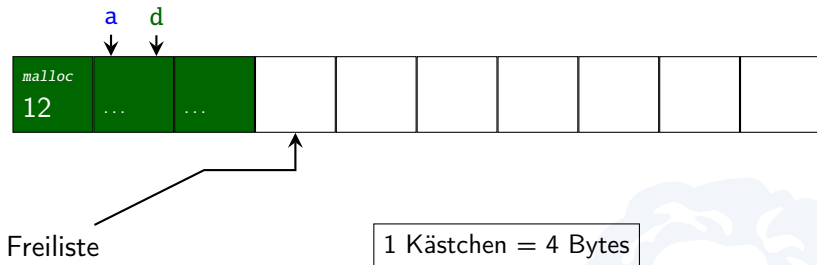
```
knoten_t *a = calloc(1, sizeof(knoten_t));  
⇒ free(a);  
double *d = calloc(1, sizeof(double));  
int *i = calloc(1, sizeof(int));  
a->nach = calloc(1, sizeof(knoten_t));
```


Speicherkorruption



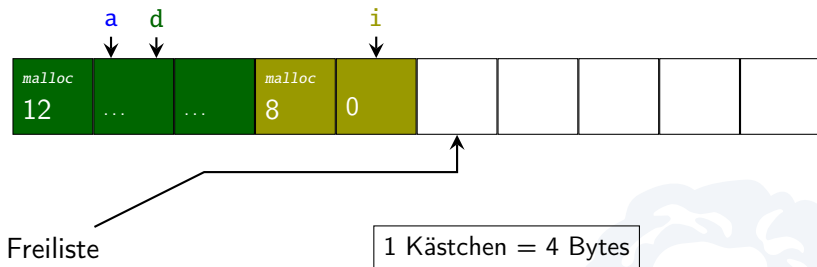
```
knoten_t *a = calloc(1, sizeof(knoten_t));  
free(a);  
⇒ double *d = calloc(1, sizeof(double));  
int *i = calloc(1, sizeof(int));  
a->nach = calloc(1, sizeof(knoten_t));
```

Speicherkorruption



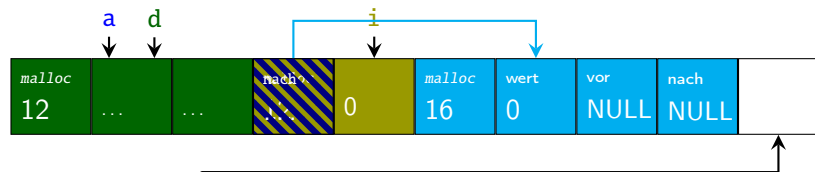
```
knoten_t *a = calloc(1, sizeof(knoten_t));  
free(a);  
double *d = calloc(1, sizeof(double));  
⇒ int *i = calloc(1, sizeof(int));  
a->nach = calloc(1, sizeof(knoten_t));
```

Speicherkorruption



```
knoten_t *a = calloc(1, sizeof(knoten_t));  
free(a);  
double *d = calloc(1, sizeof(double));  
int *i = calloc(1, sizeof(int));  
⇒ a->nach = calloc(1, sizeof(knoten_t));
```

Speicherkorruption

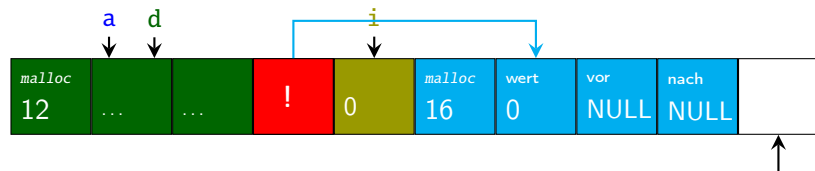


Freiliste

1 Kästchen = 4 Bytes

```
knoten_t *a = calloc(1, sizeof(knoten_t));  
free(a);  
double *d = calloc(1, sizeof(double));  
int *i = calloc(1, sizeof(int));  
a->nach = calloc(1, sizeof(knoten_t));
```

Speicherkorruption



Freiliste

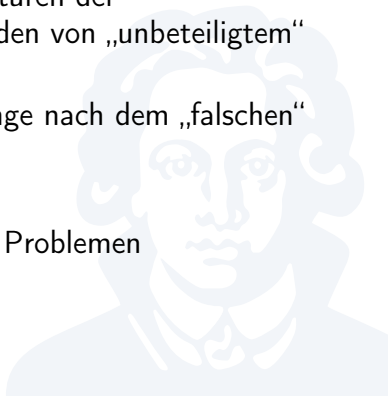
1 Kästchen = 4 Bytes

```
knoten_t *a = calloc(1, sizeof(knoten_t));  
free(a);  
double *d = calloc(1, sizeof(double));  
int *i = calloc(1, sizeof(int));  
a->nach = calloc(1, sizeof(knoten_t));
```

Speicherkorruption: Speicherinhalt beschädigt

Speicherkorruption

- Speicherkorruption:
Speicherinhalte oder Kontrollstrukturen der Speicherverwaltung (`malloc`) werden von „unbeteiligtem“ Code beschädigt
- Sichtbare Fehler treten oft erst lange nach dem „falschen“ Speicherzugriff auf
- *Sehr schwer* zu diagnostizieren
- Manifestiert sich meist in anderen Problemen



Zusammenfassung: Speicherfehler

- Entstehen durch Verletzung eines 'Vertrages':
 - Genau ein `free` pro `malloc`
 - `free`, wenn Speicher nicht mehr nötig
 - Keine Dereferenzierung von „hängenden Zeigern“
 - Zeigerarithmetik nur mit Vorsicht

Fehler	Ablage	Stapel	Statisch
Doppel- <code>free()</code>	<i>möglich</i>	nicht möglich	nicht möglich
Deallozierung vergessen	<i>möglich</i>	nicht möglich	nicht möglich
Hängender Zeiger	<i>möglich</i>	<i>möglich</i>	nicht möglich
Zeigerarithmetik-Fehler	<i>möglich</i>	<i>möglich</i>	<i>möglich</i>

Speicherfehler sind in allen Speicherformen möglich

Nächste Woche:

Der Präprozessor `cpp`
Das C-Modulsystem
Systembibliotheken

