

Überblick: Teil D Betriebssystemabstraktionen

15 Programmausführung, Nebenläufigkeit

16 Ergänzungen zur Einführung in C

17 Betriebssysteme I

18 Dateisysteme

19 Prozesse I

20 Speicherorganisation

21 Prozesse II



Speicherorganisation

```
int a;                      // a: global, uninitialized
int b = 1;                   // b: global, initialized
const int c = 2;              // c: global, const

void main() {
    static int s = 3;          // s: local, static, initialized
    int x, y;                  // x: local, auto; y: local, auto
    char* p = malloc( 100 );   // p: local, auto; *p: heap (100 byte)
}
```

Wo kommt der Speicher für diese Variablen her?

■ Statische Allokation – Reservierung beim Übersetzen / Linken

- Betrifft globale und modullokale Variablen, sowie den Code
- Allokation durch Platzierung in einer **Sektion**

.code – enthält den Programmcode

main()

.bss – enthält alle uninitialized / mit 0 initialisierten Variablen

a

.data – enthält alle initialisierten Variablen

b,s

.rodata – enthält alle initialisierten unveränderlichen Variablen

c

■ Dynamische Allokation – Reservierung zur Laufzeit

- Betrifft lokale Variablen und explizit angeforderten Speicher

Stack – enthält alle **aktuell gültigen** lokalen Variablen

x,y,p

Heap – enthält explizit mit **malloc()** angeforderte Speicherbereiche

*p



Speicherorganisation auf einem μ C

```
int a;                      // a: global, uninitialized
int b = 1;                   // b: global, initialized
const int c = 2;              // c: global, const

void main() {
    static int s = 3;          // s: local, static, initialized
    int x, y;                  // x: local, auto; y: local, auto
    char* p = malloc( 100 );   // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm

Symbol Table <a>	
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

ELF-Binary

Beim Übersetzen und Linken werden die Programmelemente in entsprechenden Sektionen der ELF-Datei zusammen gefasst. Informationen zur Größe der .bss-Sektion landen ebenfalls in .rodata.



Speicherorganisation auf einem μ C

```
int a;                      // a: global, uninitialized
int b = 1;                   // b: global, initialized
const int c = 2;              // c: global, const

void main() {
    static int s = 3;          // s: local, static, initialized
    int x, y;                  // x: local, auto; y: local, auto
    char* p = malloc( 100 );   // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm

Flash / ROM

.data	s=3 b=1
.rodata	c=2
.text	main

Symbol Table <a>	
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

Zur Installation auf dem μ C werden .text und .[ro]data in den Flash-Speicher des μ C geladen.

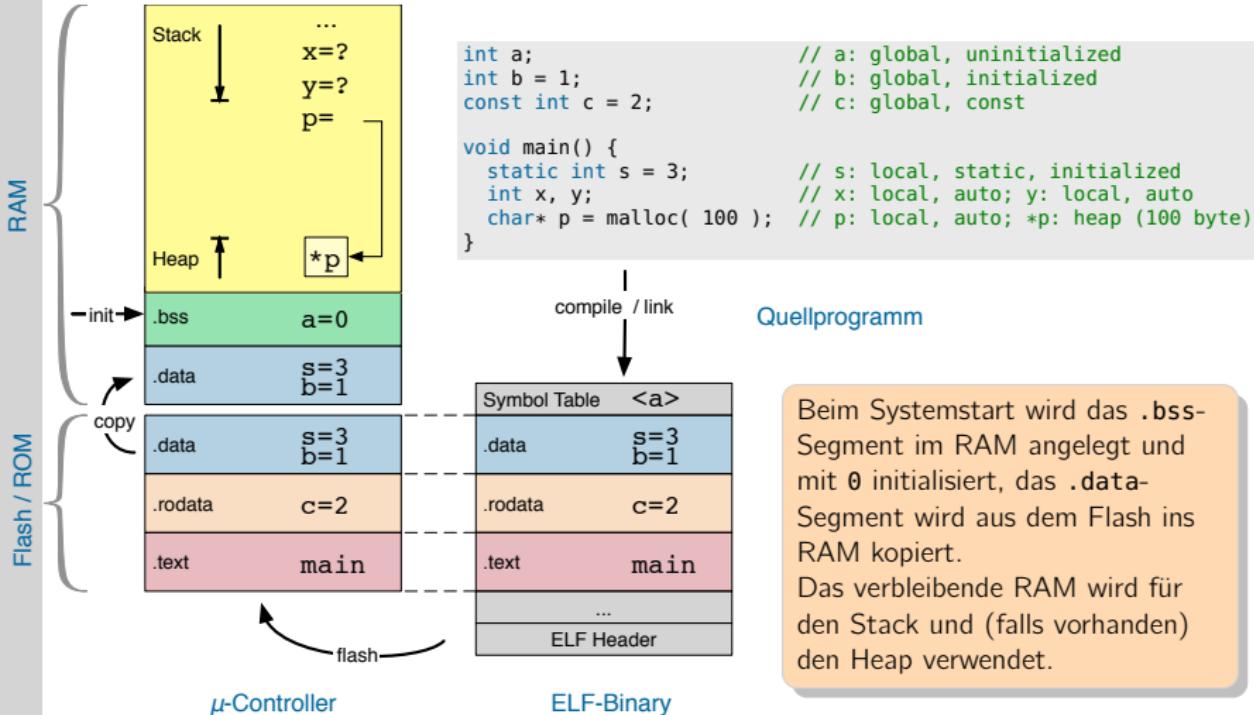
flash

μ -Controller

ELF-Binary



Speicherorganisation auf einem μ C



Verfügt die Architektur über keinen Daten-Flashspeicher (beim ATmega der Fall → 14-3), so werden konstante Variablen ebenfalls in `.data` abgelegt (und belegen zur Laufzeit RAM).



- **Programm:** Folge von Anweisungen
- **Prozess:** Betriebssystemkonzept zur Ausführung von Programmen
 - Programm, das sich in Ausführung befindet, und seine Daten
(Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
 - Eine konkrete **Ausführungsumgebung** für ein Programm (Prozessor, **Speicher**, ...) → vom Betriebssystem verwalteter **virtueller Computer**
- Jeder Prozess bekommt einen **virtuellen Adressraum** zugeteilt
 - 4 GB auf einem 32-Bit-System, davon bis zu 3 GB für die Anwendung
 - In das verbleibende GB werden Betriebssystem und *memory-mapped* Hardware (z. B. PCI-Geräte) eingeblendet
 - Daten des Betriebssystems werden durch Zugriffsrechte geschützt
 - Zugriff auf andere Prozesse ist nur über das Betriebssystem möglich
 - Virtueller Speicher wird durch das Betriebssystem auf physikalischen (Hintergrund-)Speicher abgebildet



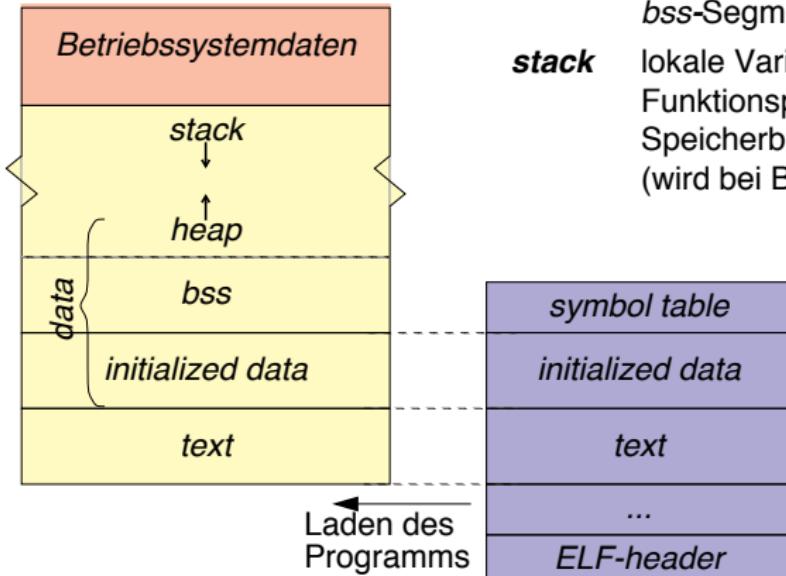
Speicherorganisation in einem UNIX-Prozess (Forts.)

text Programmcode
data globale und static Variablen

bss nicht initialisierte globale und *static* Variablen (wird vor der Vergabe an den Prozess mit 0 vorbelegt)

heap dynamische Erweiterungen des *bss*-Segments (*sbrk(2)*, *malloc(3)*)

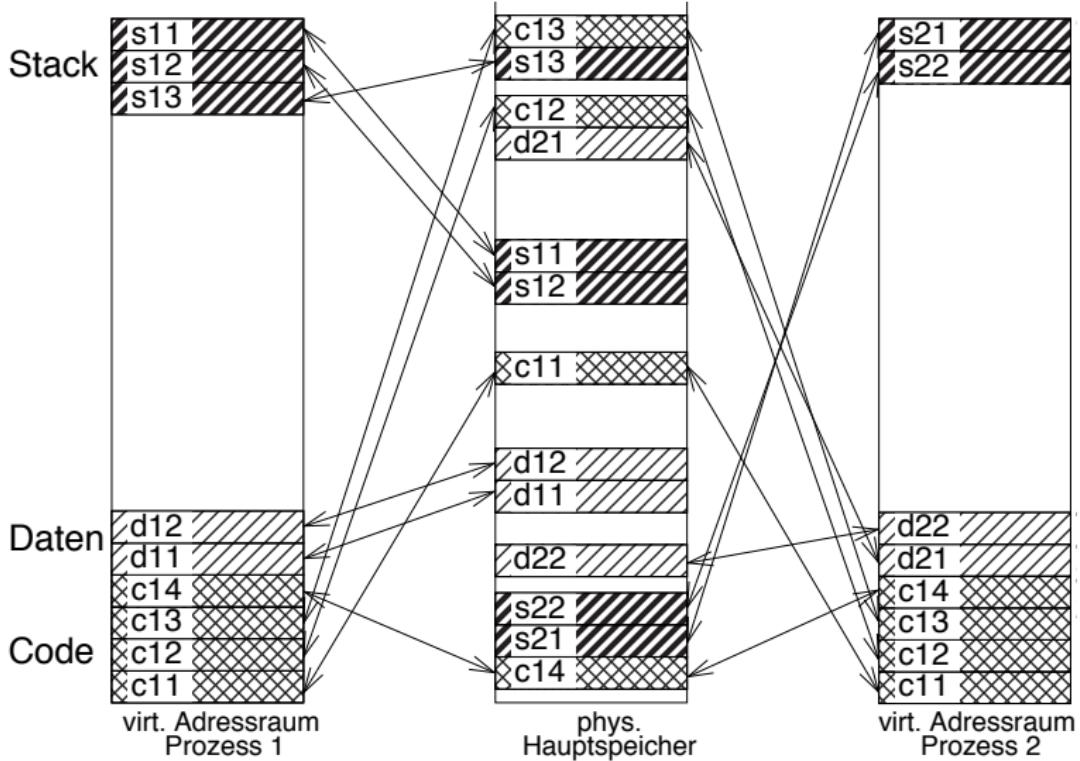
stack lokale Variablen, Funktionsparameter, Speicherbereiche für Registerinhalte, (wird bei Bedarf dynamisch erweitert)

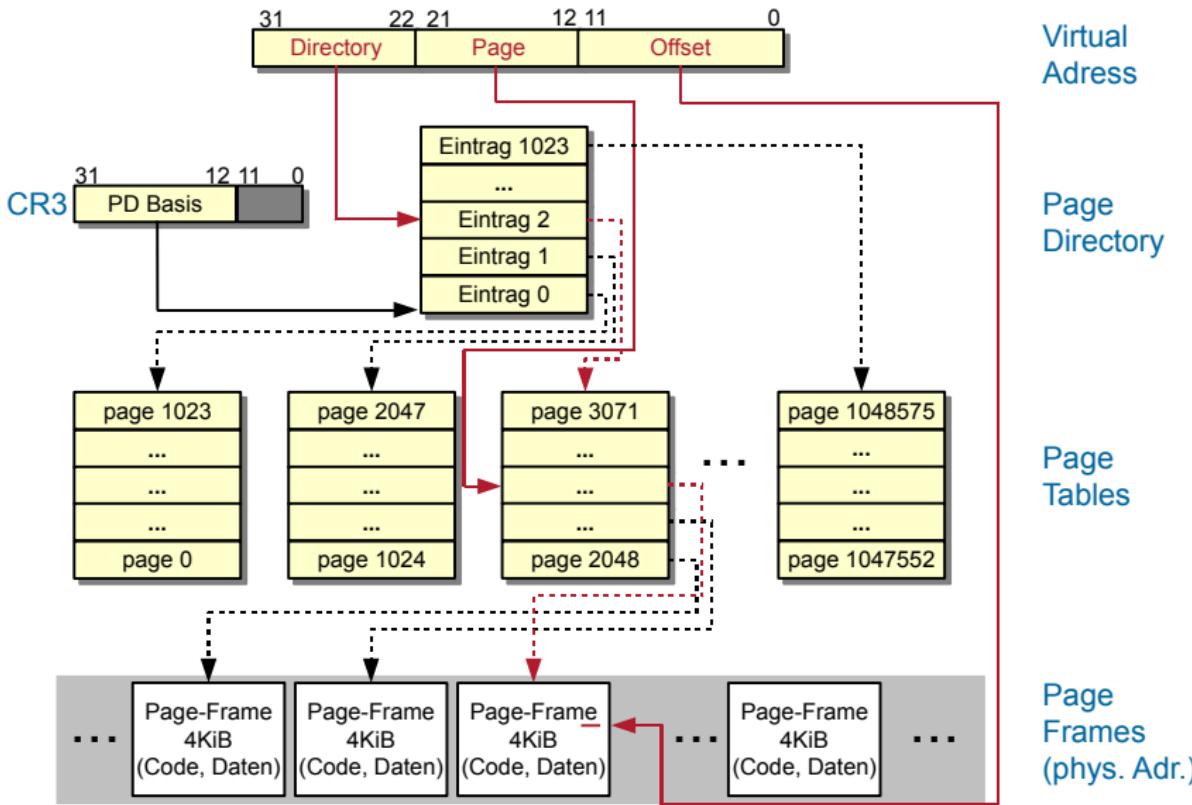


Seitenbasierte Speicherverwaltung

- Die Abbildung von virtuellem Speicher (*VS*) auf physikalischen Speicher (*PS*) erfolgt durch **Seitenaddressierung (Paging)**
 - *VS* eines Prozesses ist unterteilt in **Speicherseiten (Memory Pages)**
 - kleine Adressblöcke, üblich sind z. B. 4 KiB und 4 MiB Seiten
 - in dieser Granularität wird Speicher **vom Betriebssystem** zugewiesen
 - *PS* ist analog unterteilt in **Speicherrahmen (Page Frames)**
 - Abbildung: *Seite* \mapsto *Rahmen* über eine **Seitentabelle (Page Table)**
 - Umrechnung *VS* auf *PS* bei jedem Speicherzugriff
 - Hardwareunterstützung durch **MMU (Memory Management Unit)**
 - Betriebssystem kann Seiten auf den Hintergrundspeicher auslagern
 - Abbildung ist nicht linkseindeutig: Seiten aus mehreren Prozessen können auf denselben Rahmen verweisen (z. B. gemeinsamer Programmcode)
- Seitenbasierte Speicherverwaltung ist auch ein **Schutzkonzept**
 - Seiten sind mit Zugriffsrechten versehen: *Read*, *Read–Write*, *Execute*
 - MMU überprüft bei der Umrechnung, ob der Zugriff erlaubt ist







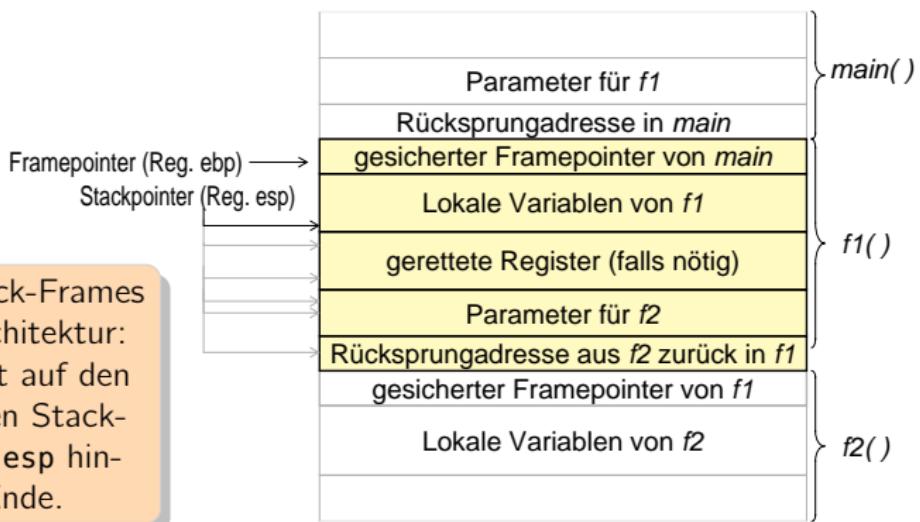
Dynamische Speicherallokation: Heap

- **Heap** := Vom Programm explizit verwalteter RAM-Speicher
 - Lebensdauer ist unabhängig von der Programmstruktur
- Anforderung und Wiederfreigabe über zwei Basisoperationen
 - `void* malloc(size_t n)` fordert einen Speicherblock der Größe n an; Rückgabe bei Fehler: 0-Zeiger (`NULL`)
 - `void free(void* pmem)` gibt einen zuvor mit `malloc()` angeforderten Speicherblock vollständig wieder frei
- Beispiel

```
#include <stdlib.h>
int* intArray( uint16_t n ) {      // alloc int[n] array
    return (int*) malloc( n * sizeof int );
}
void main() {
    int* array = intArray(100);      // alloc memory for 100 ints
    if( array ) {                  // malloc() returns NULL on failure
        ...                         // if succeeded, use array
        array[99] = 4711;
        ...
        free( array );             // free allocated block (** IMPORTANT! **)
    }
}
```



- Lokale Variablen, Funktionsparameter und Rücksprungadressen werden vom Übersetzer auf dem **Stack** (Stapel, Keller) verwaltet
 - Prozessorregister [e]sp zeigt immer auf den nächsten freien Eintrag
 - Stack „wächst“ (architekturabhängig) „von oben nach unten“
- Die Verwaltung erfolgt in Form von **Stack-Frames**



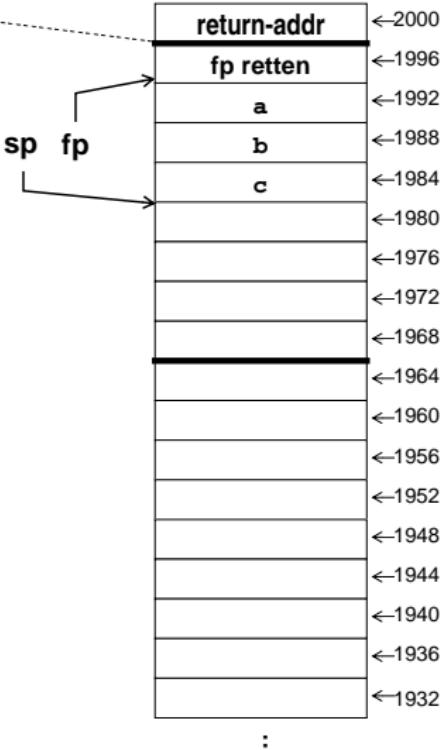
Aufbau eines Stack-Frames auf der IA-32-Architektur:
Register ebp zeigt auf den Beginn des aktiven Stack-Frames; Register esp hinter das aktuelle Ende.



Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
  
    return(a);  
}
```

Stack-Frame für
main erstellen
 $\&a = fp-4$
 $\&b = fp-8$
 $\&c = fp-12$



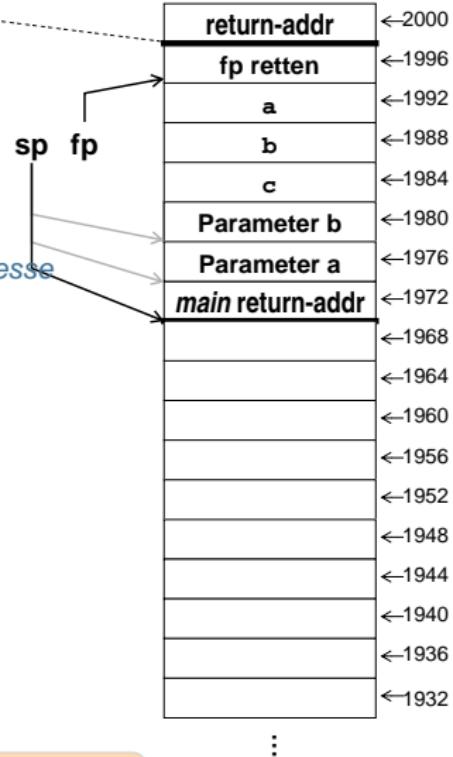
Beispiel hier für 32-Bit-Architektur (4-Byte `ints`), `main()` wurde soeben betreten



Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
  
    return(a);  
}
```

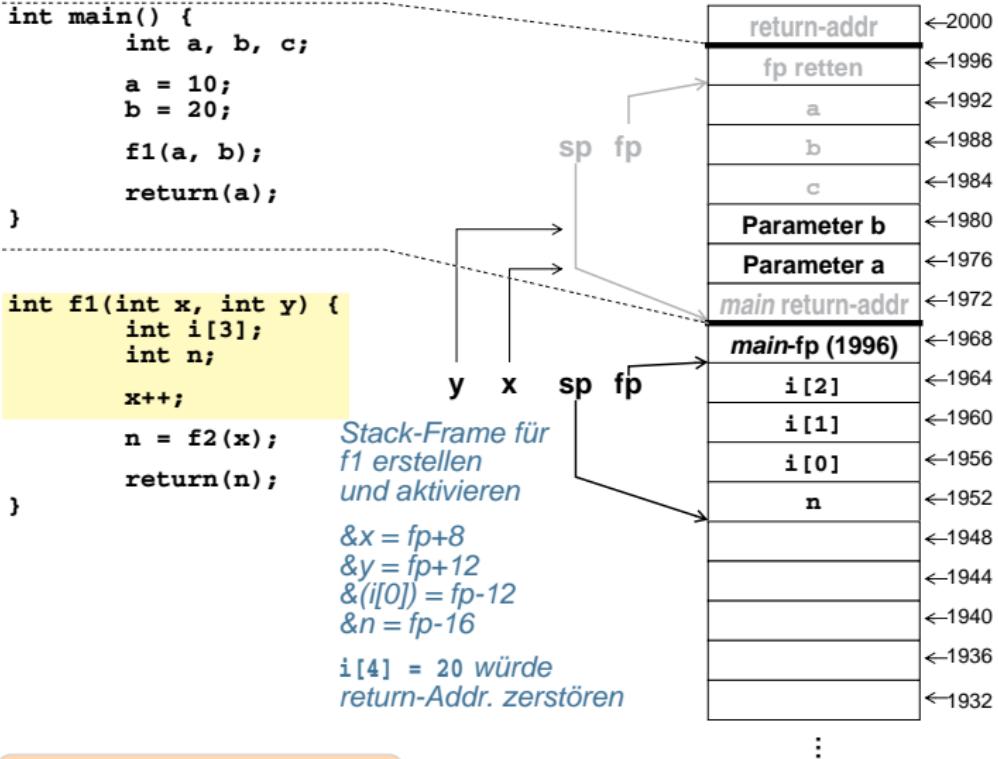
Parameter
auf Stack legen
Bei Aufruf
Rückprungadresse
auf Stack legen



main() bereitet den Aufruf von f1(int, int) vor



Stack-Aufbau bei Funktionsaufrufen



f1() wurde soeben betreten



Stack-Aufbau bei Funktionsaufrufen

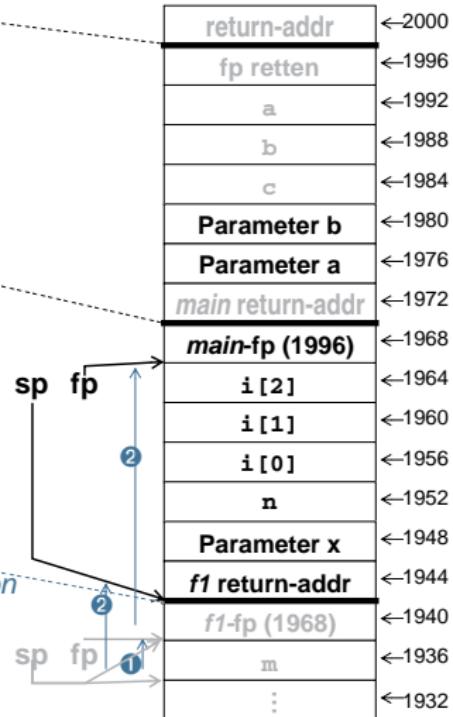
```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
  
    return(a);  
}
```

```
int f1(int x, int y) {  
    int i[3];  
    int n;  
  
    x++;  
  
    n = f2(x);  
  
    return(n);  
}
```

```
int f2(int z) {  
    int m;  
    m = 100;  
  
    return(z+1);  
}
```

Stack-Frame von
f2 abräumen

① $sp = fp$
② $fp = pop(sp)$



f2() bereitet die Terminierung vor (wurde von f1() aufgerufen und ausgeführt)

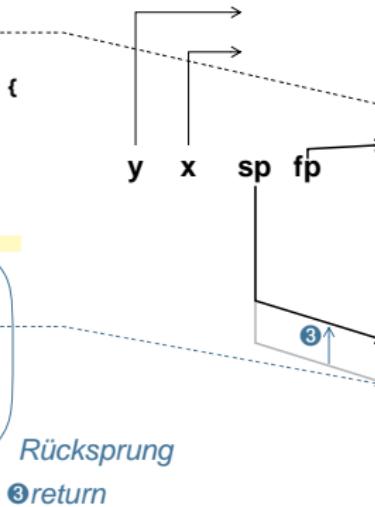


Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
  
    a = 10;  
    b = 20;  
  
    f1(a, b);  
  
    return(a);  
}
```

```
int f1(int x, int y) {  
    int i[3];  
    int n;  
  
    x++;  
  
    n = f2(x);  
    return(n);  
}
```

```
int f2(int z) {  
    int m;  
  
    m = 100;  
    return(z+1);  
}
```

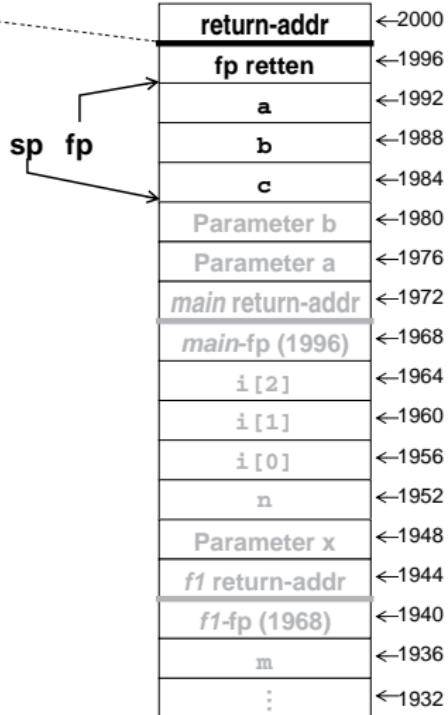


`f2()` wird verlassen



Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);
```



zurück in `main()`

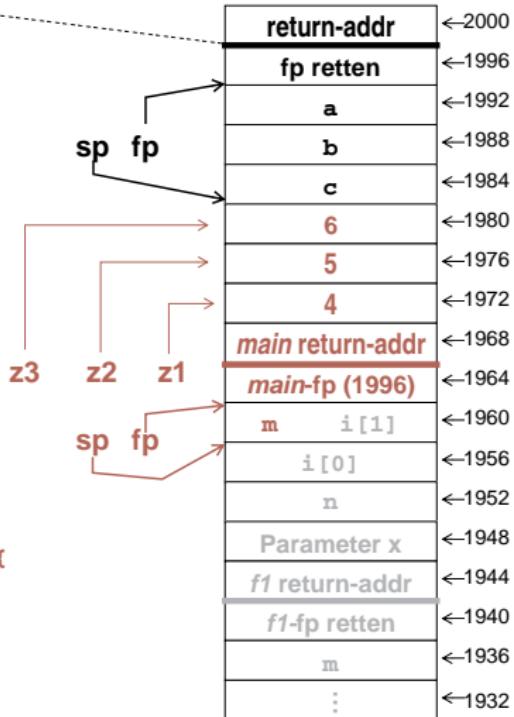


Stack-Aufbau bei Funktionsaufrufen

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    f3(4,5,6);  
}
```

*was wäre, wenn man nach
f1 jetzt eine Funktion f3
aufrufen würde?*

```
int f3(int z1, int z2, int z3) {  
    int m;  
  
    return(m);  
}
```



m wird nicht initialisiert ↗ „erbt“ alten Wert vom Stapel



Statische versus dynamische Allokation

- Bei der **μ C-Entwicklung** wird **statische Allokation** bevorzugt
 - **Vorteil:** Speicherplatzbedarf ist bereits nach dem Übersetzen / Linken exakt bekannt (kann z. B. mit `size` ausgegeben werden)
 - Speicherprobleme frühzeitig erkennbar (Speicher ist knapp! ↪ 1-3)

```
lohmann@faui48a:$ size sections.avr
text      data      bss      dec      hex filename
682        10        6     698      2ba sections.avr
```

Sektionsgrößen des
Programms von ↪ 20-1

- ~ Speicher möglichst durch **static**-Variablen anfordern
 - Regel der geringstmöglichen Sichtbarkeit beachten ↪ 10-4
 - Regel der geringstmöglichen Lebensdauer „sinnvoll“ anwenden
- Ein Heap ist **verhältnismäßig teuer** ~ wird möglichst vermieden
 - Zusätzliche Specherkosten durch Verwaltungsstrukturen und Code
 - Speicherbedarf zur Laufzeit schlecht abschätzbar
 - Risiko von Programmierfehlern und Speicherlecks

