J Programme, Prozesse und Speicher

- **Programm:** Folge von Anweisungen
  (hinterlegt beispielsweise als ausführbare Datei auf dem Hintergrundspeicher)

- **Prozess:** Betriebssystemkonzept
  - ➤ Programm, das sich in Ausführung befindet, und seine Daten
    (Beachte: ein Programm kann sich mehrfach in Ausführung befinden)
  - ➤ eine konkrete Ausführungsumgebung für ein Programm
    Speicher, Rechte, Verwaltungsinformation (verbrauchte Rechenzeit,...),...

- jeder Prozess bekommt einen eigenen virtuellen Adressraum zur Verfügung gestellt
  - ➤ eigener (virtueller) Speicherbereich von 0 bis 2 GB (oder mehr bis 4 GB)
  - ➤ Datenbereichevon verschiedenen Prozessen und Betriebssystem sind gegeneinander geschützt
  - ➤ Datentransfer zwischen Prozessen nur durch Vermittlung des Betriebssystems möglich

J.1 Speicherorganisation eines Prozesses

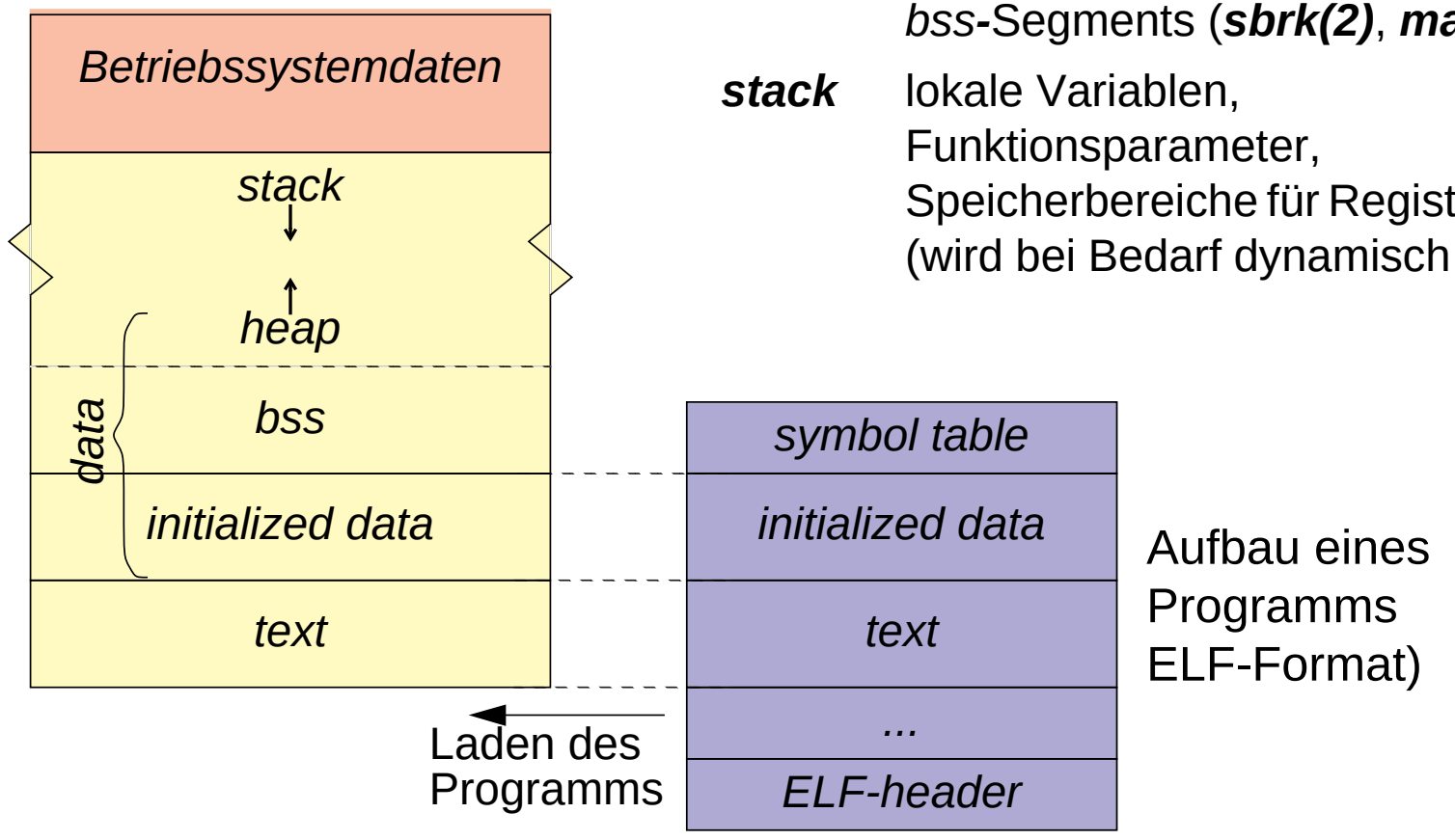
text Programmkode

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)

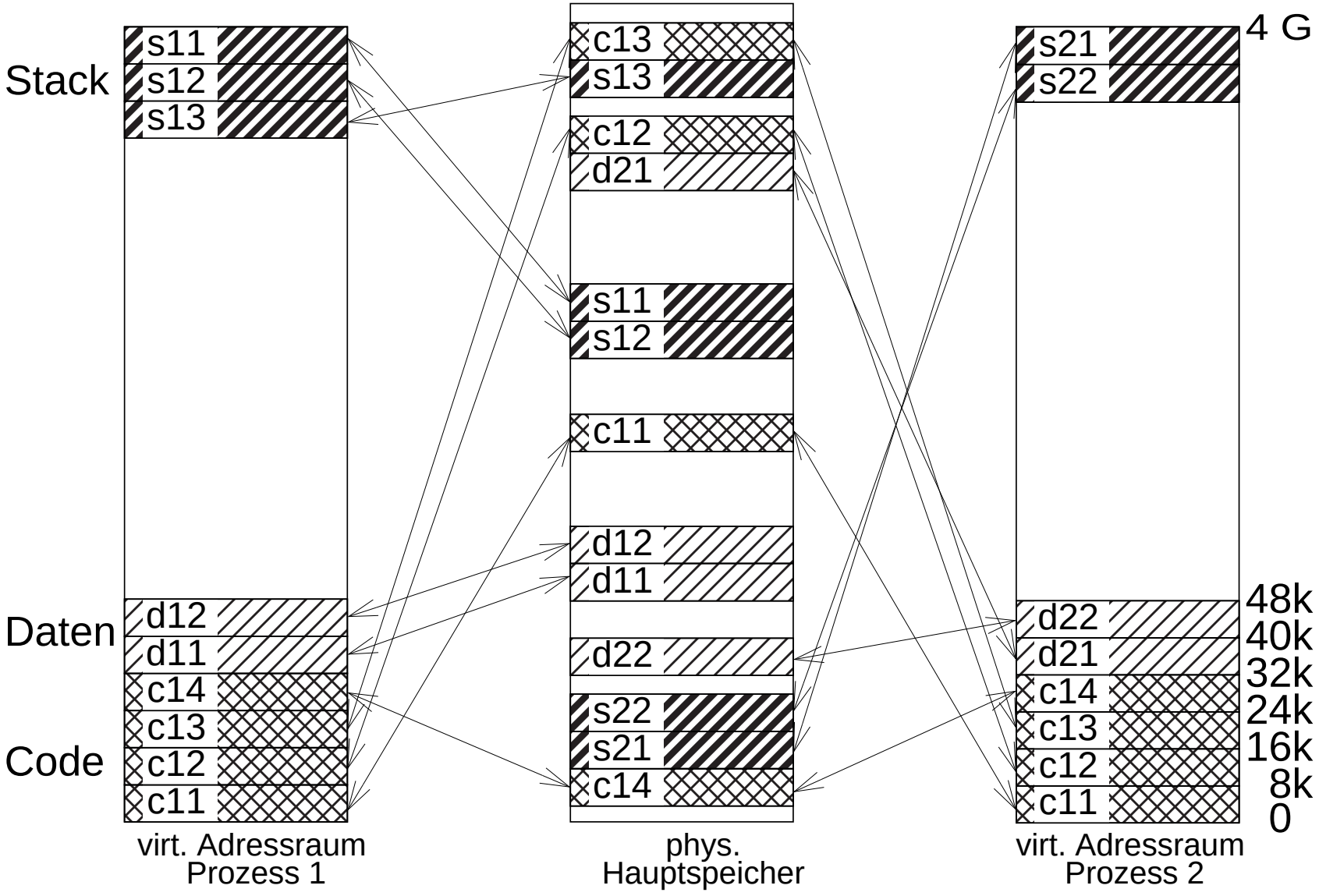


Aufbau eines Programms (ELF-Format)

J.1 Speicherorganisation eines Prozesses (2)

- Abbildung des virtuellen Adressraums in den realen Hauptspeicher durch Seitenadressierung (**Paging)**
  - ➤ Adreßraum ist in kleine (4 oder 8 kB) Stücke unterteilt (**Seiten**)
  - ➤ jede Seite wird über eine Tabelle in ein entsprechendes Stück des Hauptspeichers (**Kachel**) abgebildet
  - ➤ bei jedem Speicherzugriff wird die virtuelle Adresse in die entsprechende physikalische Adresse umgerechnet
    (spezielle Hardware: Memory Management Unit - MMU)
  - ➤ zu jeder Seite sind Zugriffsrechte vermerkt (nur lesen, lesen+schreiben, Maschinenbefehle ausführen)
  - ➤ eine Seite kann bei Speichermangel von Betriebssystem auf Festplatte ausgelagert werden und bei Bedarf automatisch wieder eingelagert werden

J.1 Speicherorganisation eines Prozesses(3)



J.2 Stackaufbau eines Prozesses

1 Prinzip

- für jede Funktion wird ein ***Stack-Frame*** angelegt, in dem
  - lokale Variablen der Funktion
  - Aufrufparameter an weitere Funktionen
  - Registerbelegung der Funktion während des Aufrufs weiterer Funktionen

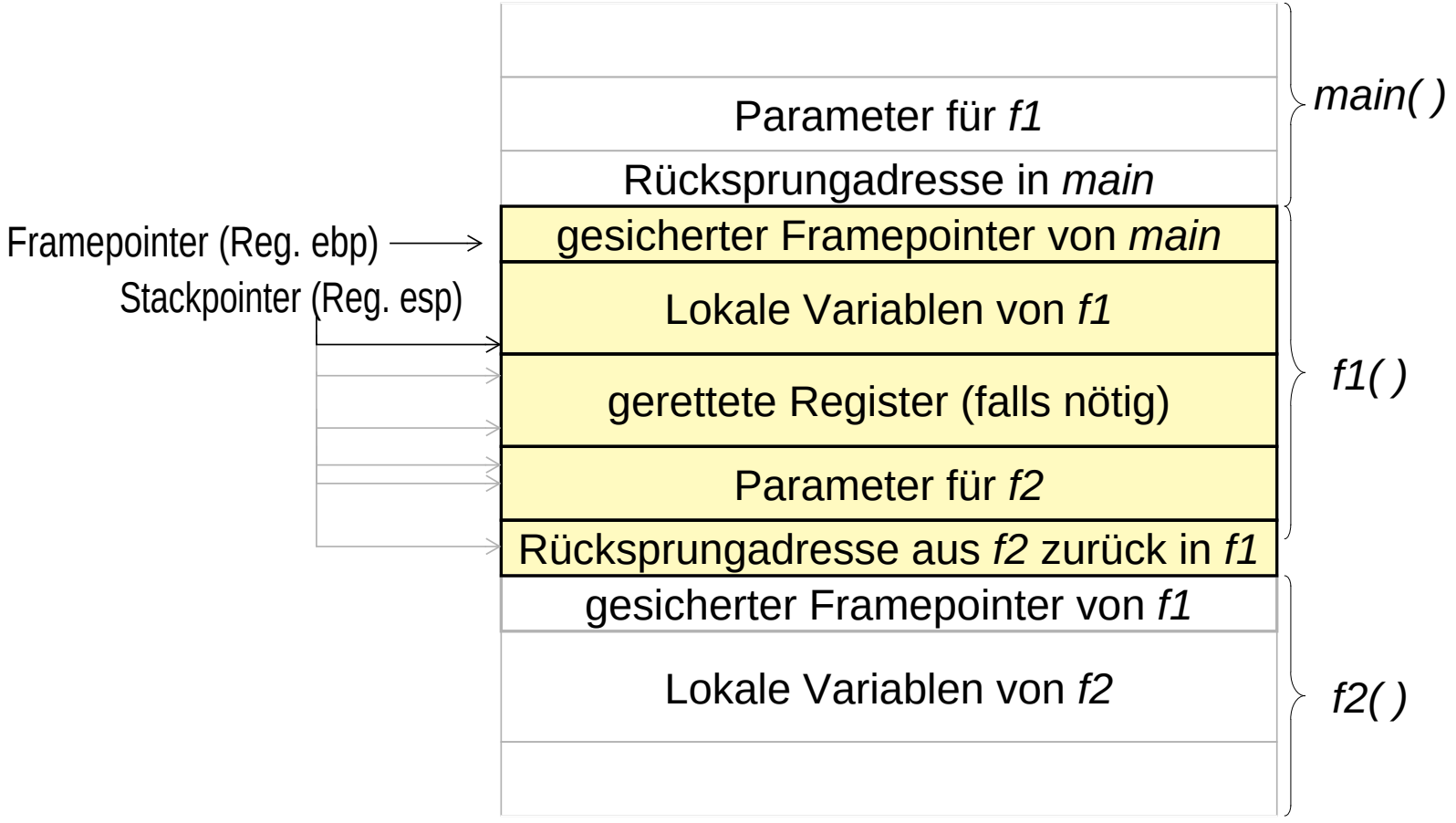
  gespeichert werden

- Stackorganisation ist abhängig von
  - Prozessor
  - Compiler und
  - Betriebssystem

- Beispiele aus einem UNIX auf Intel-Prozessor (typisch für CISC)
  - RISC-Prozessoren mit Registerfiles gehen teilweise anders vor!

2 Beispiel

- Aufbau eines **Stack-Frames** (Funktionen *main( ), f1( ), f2( )*)

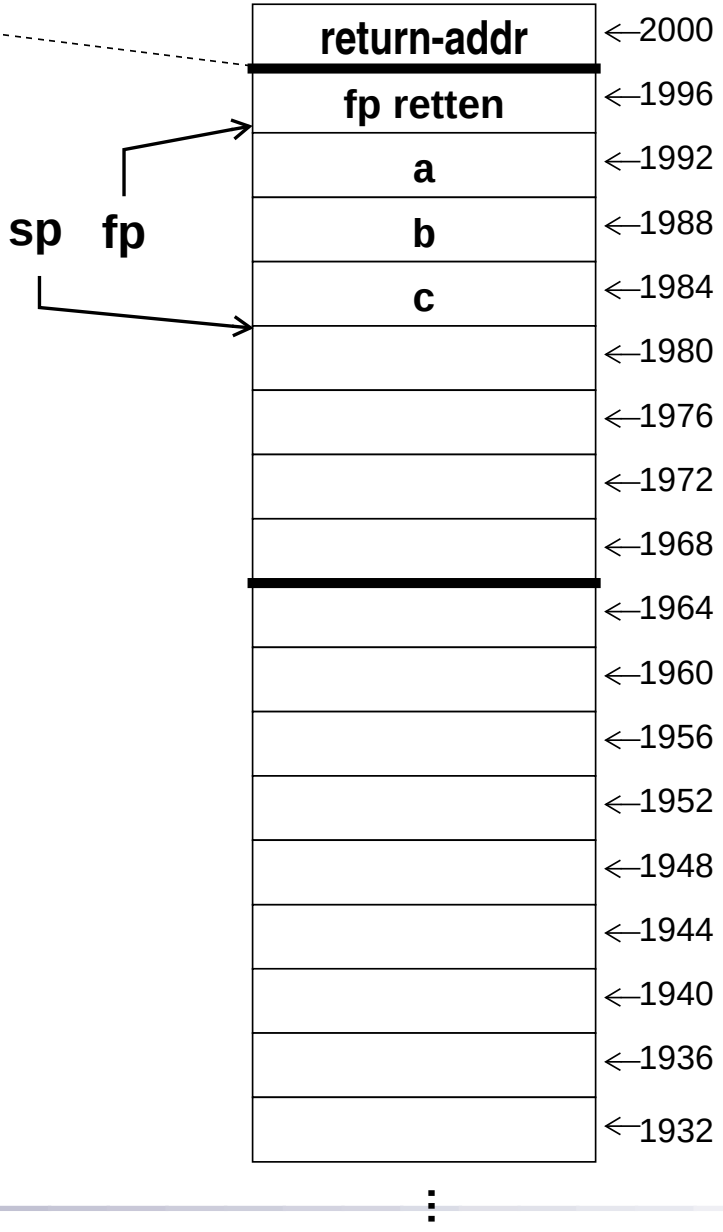


➤ Achtung: architekturabängige Optimierungen können zu Padding (Füllbytes) führen!

2 Stack mehrerer Funktionsaufrufe

```
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

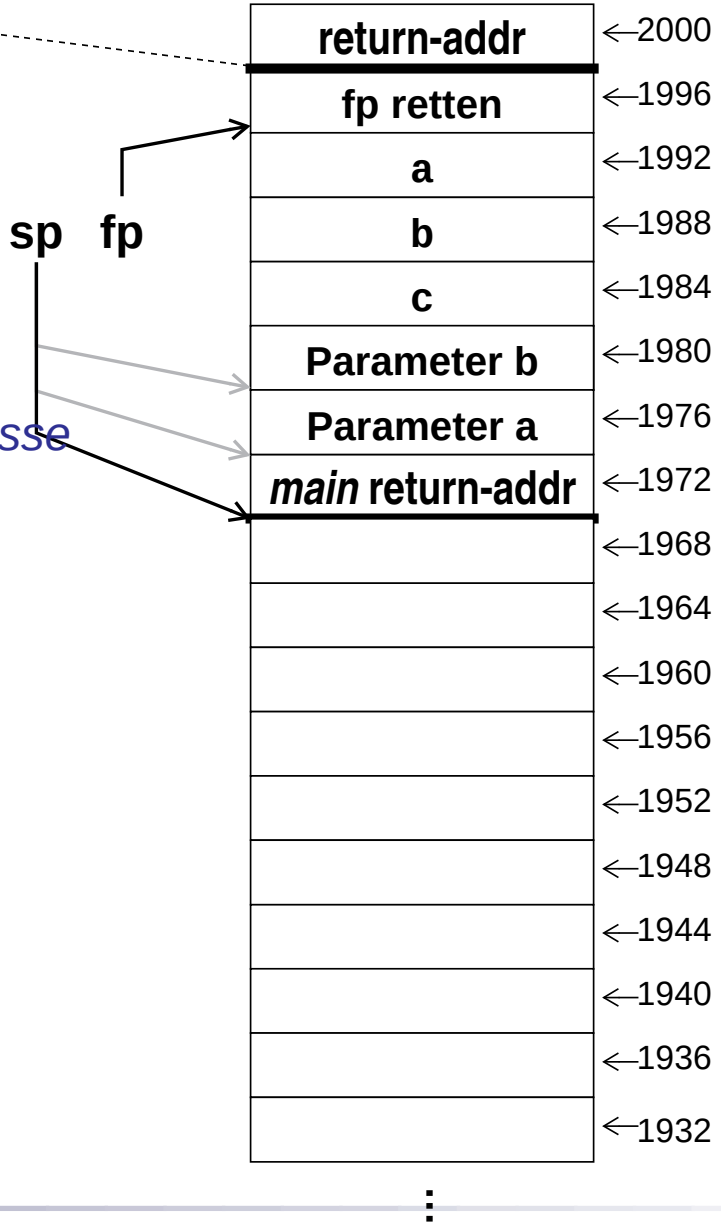


2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

Parameter auf Stack legen

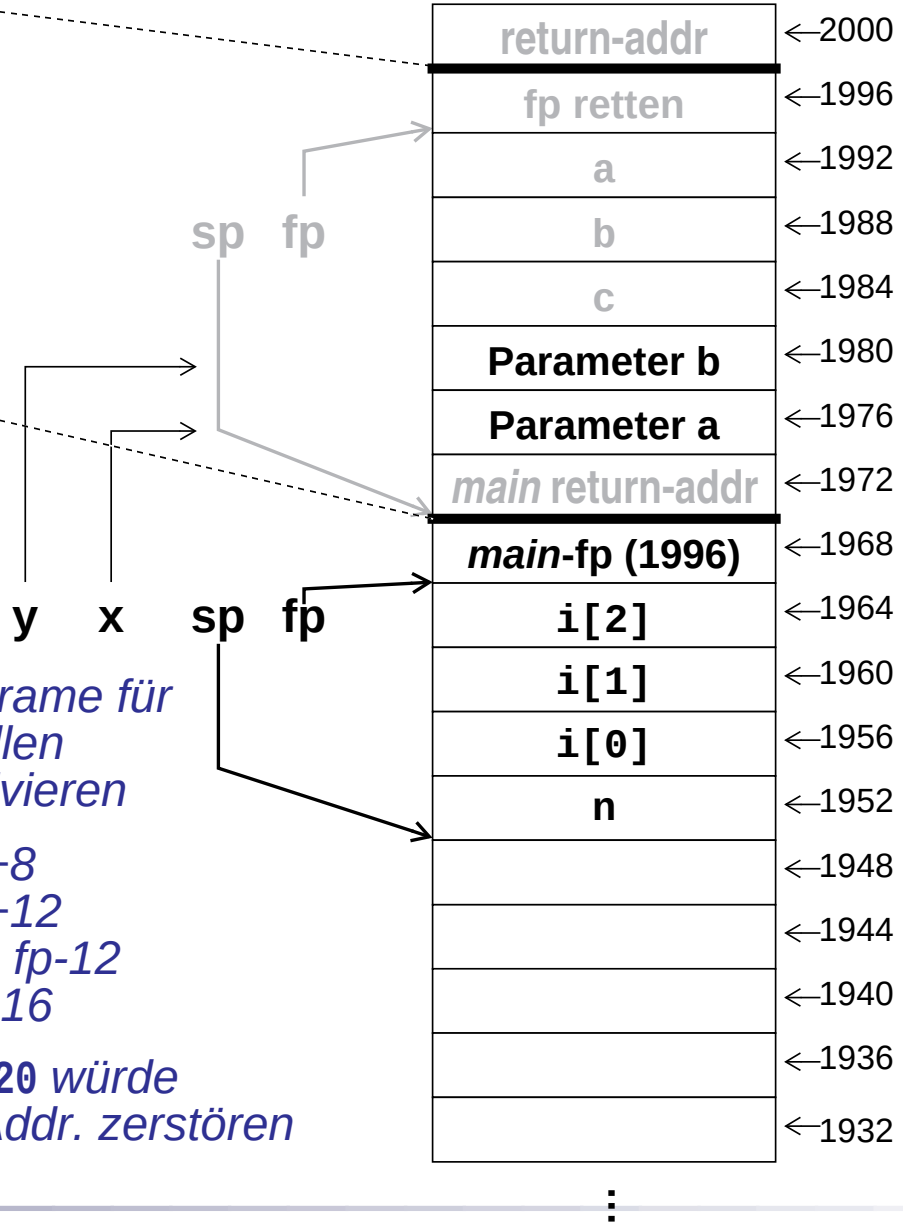
Bei Aufruf Rücksprungadresse auf Stack legen



2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



Stack-Frame für f1 erstellen und aktivieren

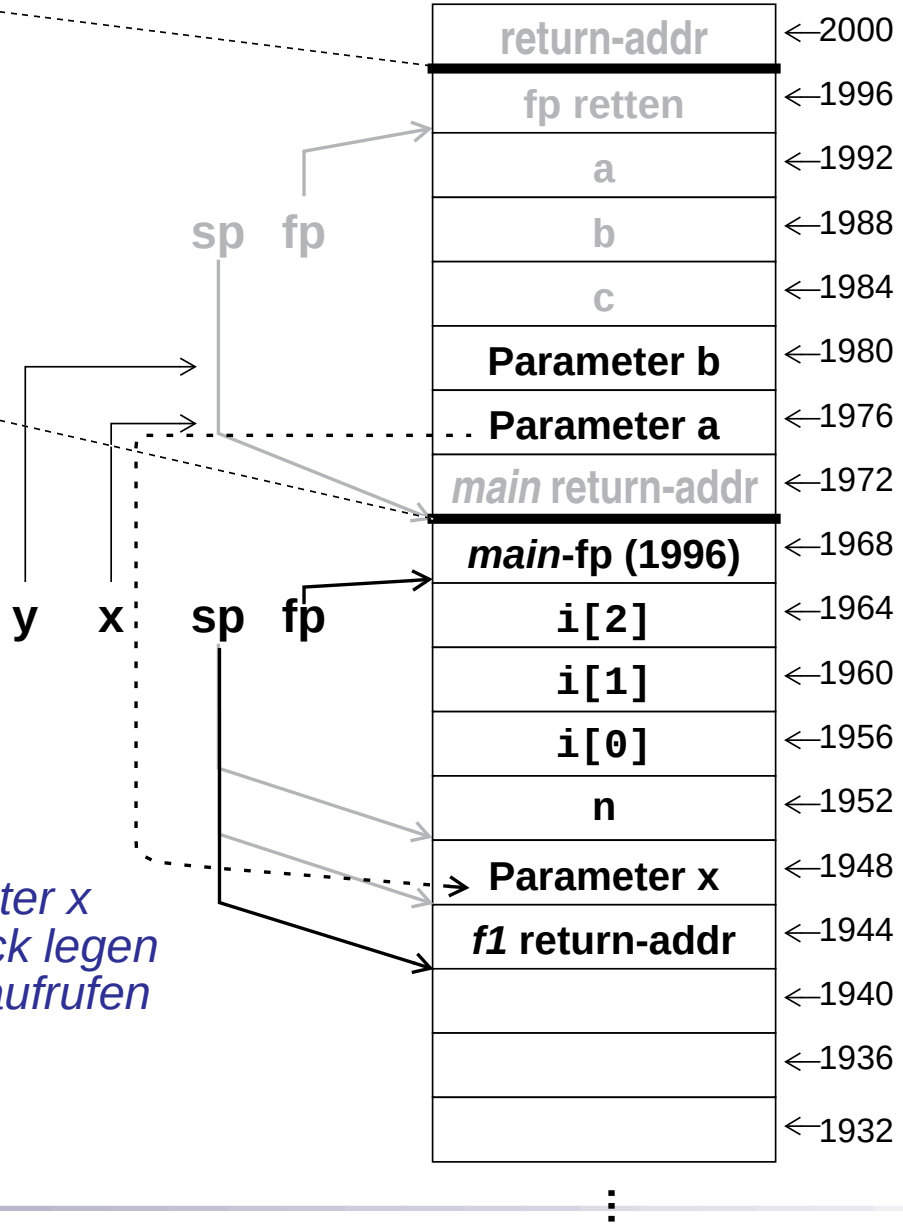
&x = fp+8
&y = fp+12
&(i[0]) = fp-12
&n = fp-16

`i[4] = 20` *würde return-Addr. zerstören*

2 Stack mehrerer Funktionsaufrufe

```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```

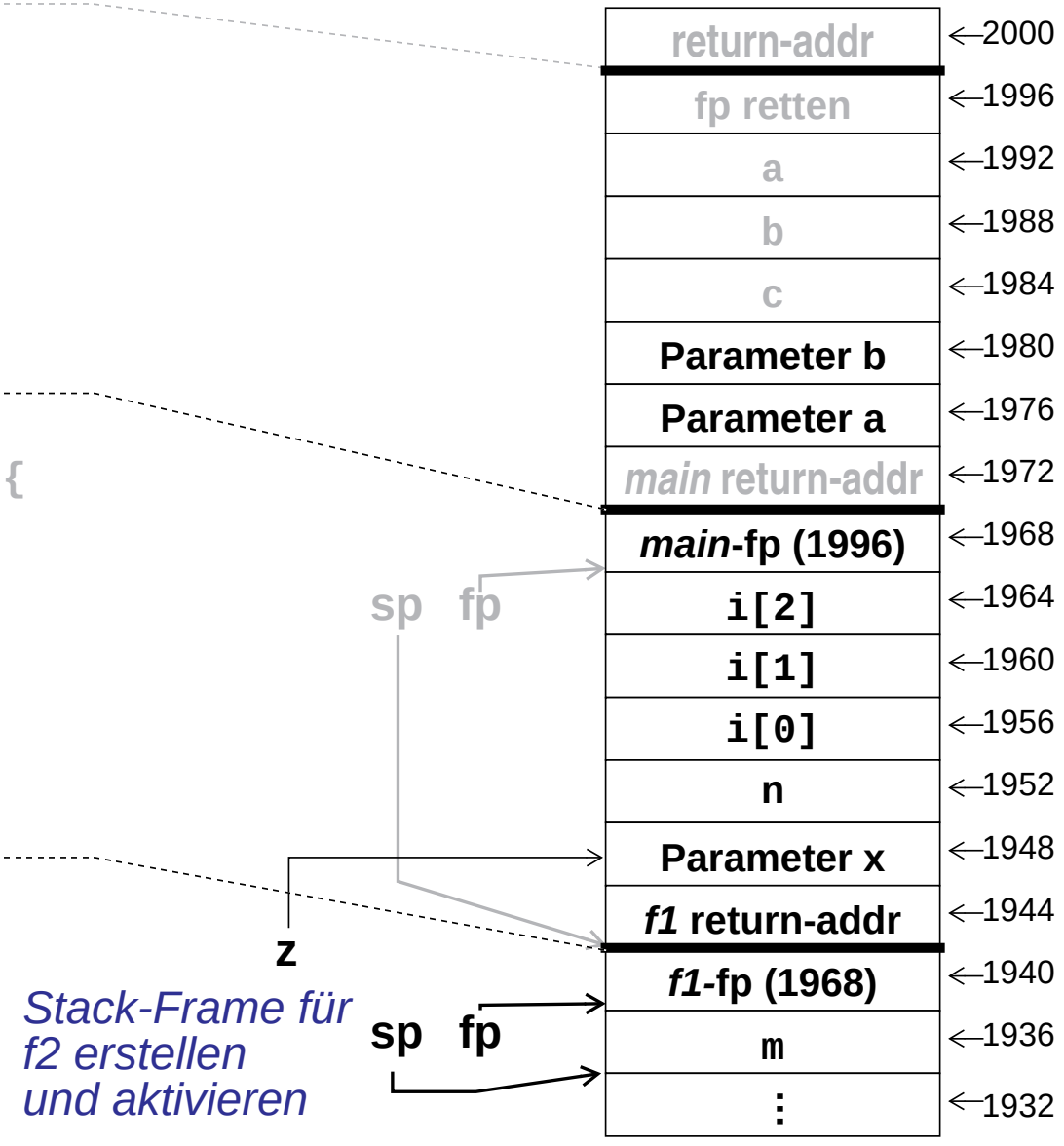
```
int f1(int x, int y) {
        int i[3];
        int n;

        x++;

        n = f2(x);

        return(n);
}
```



Parameter x auf Stack legen und f2 aufrufen

2 Stack mehrerer Funktionsaufrufe

```
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);
}
```

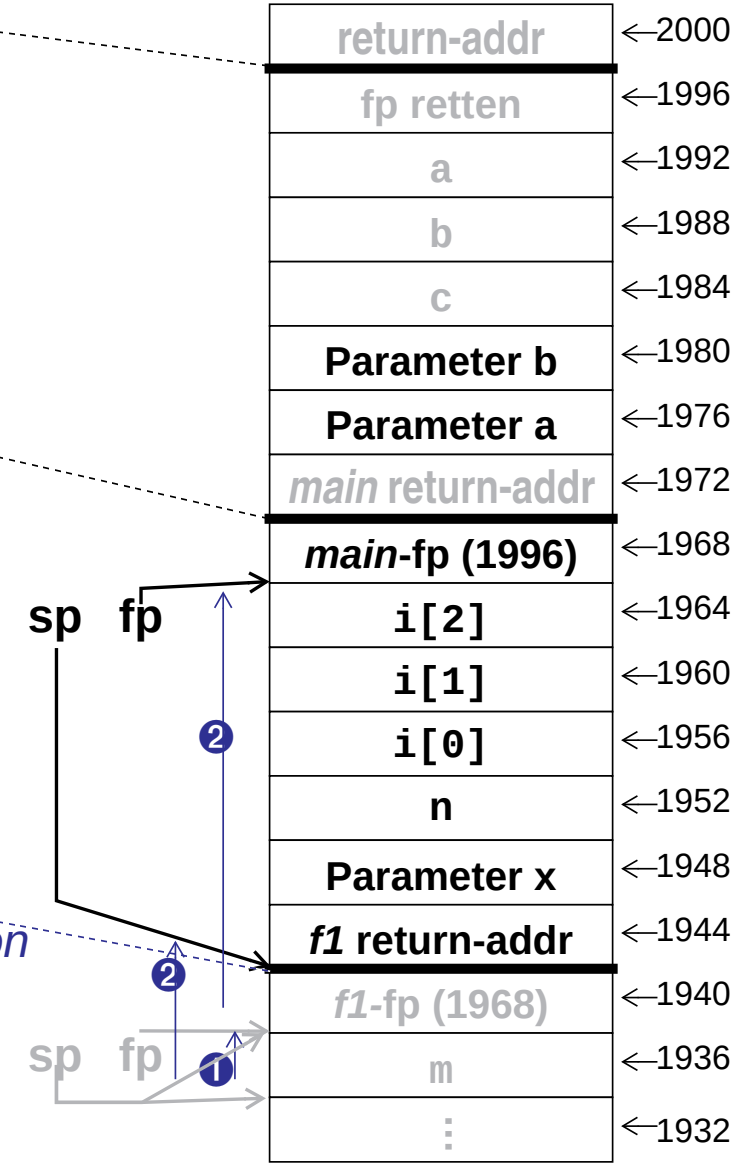


2 Stack mehrerer Funktionsaufrufe

```
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);
}
```

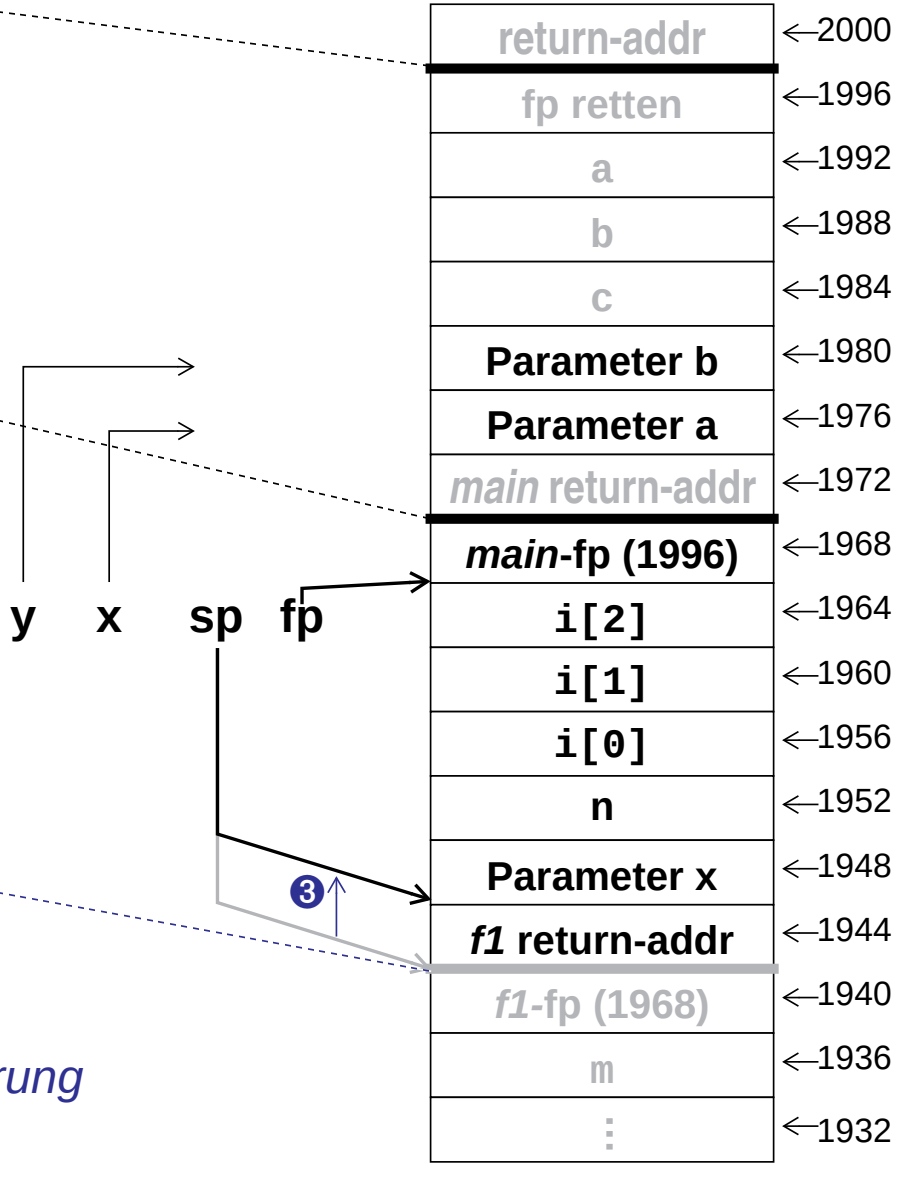
Stack-Frame von f2 abräumen

❶ *sp = fp*
❷ *fp = pop(sp)*



2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



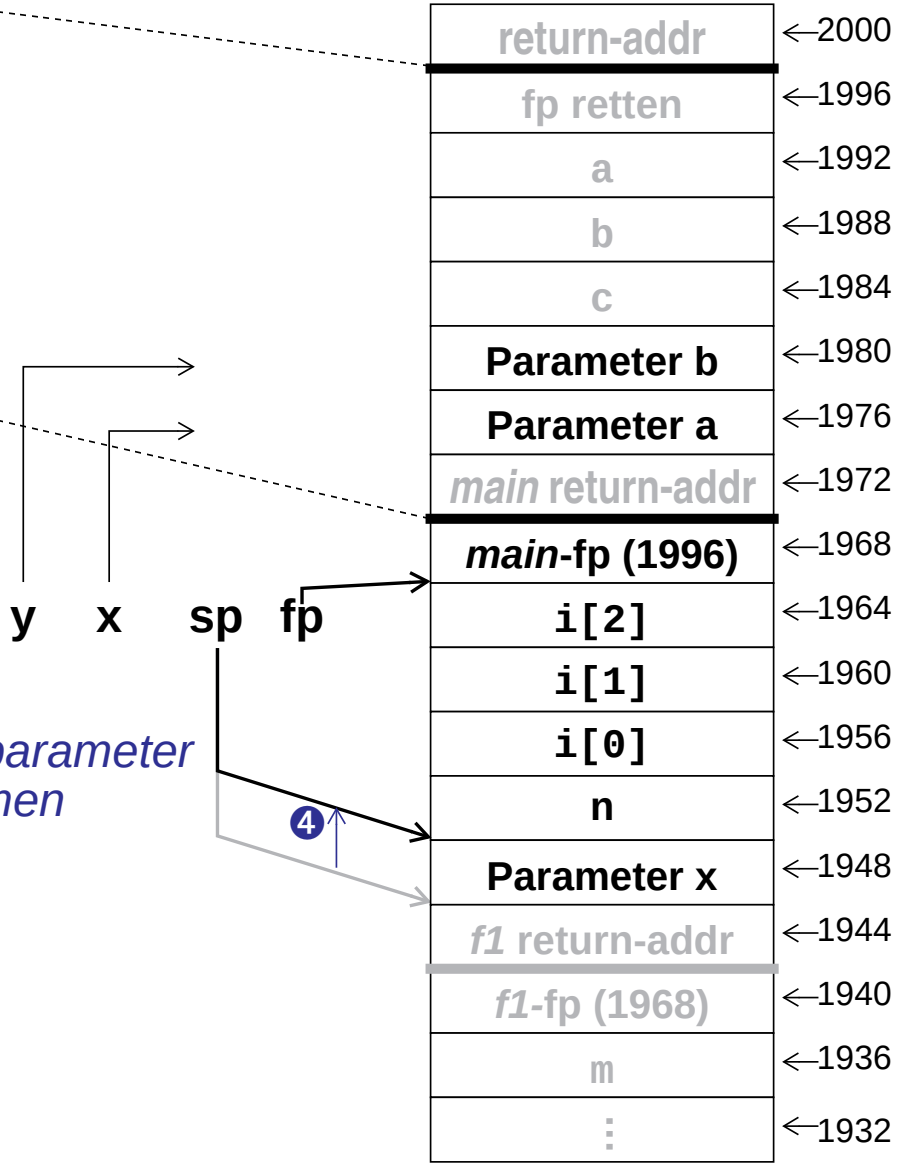
Rücksprung

❸return

2 Stack mehrerer Funktionsaufrufe

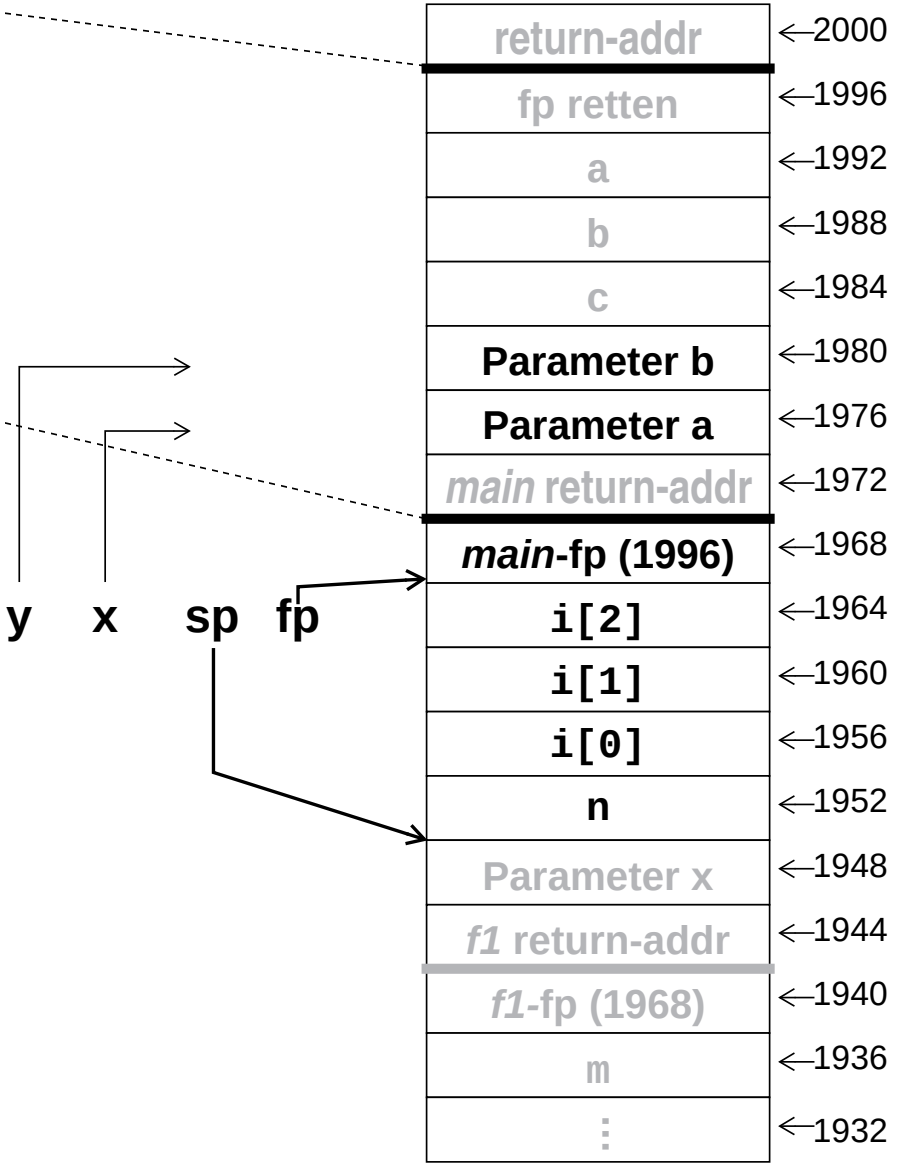
```
int main() {
        int a, b, c;

        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);
}
```

❹ *Aufrufparameter abräumen*



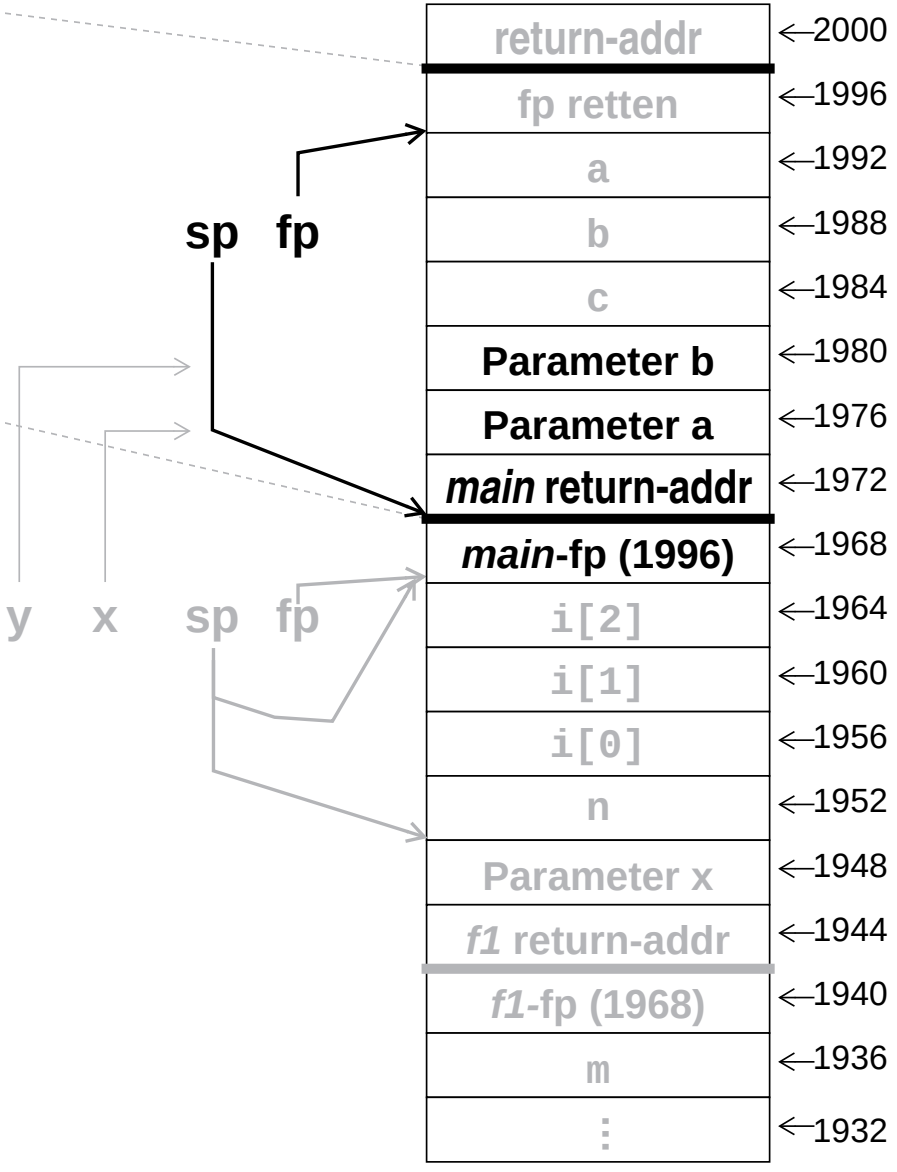
2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



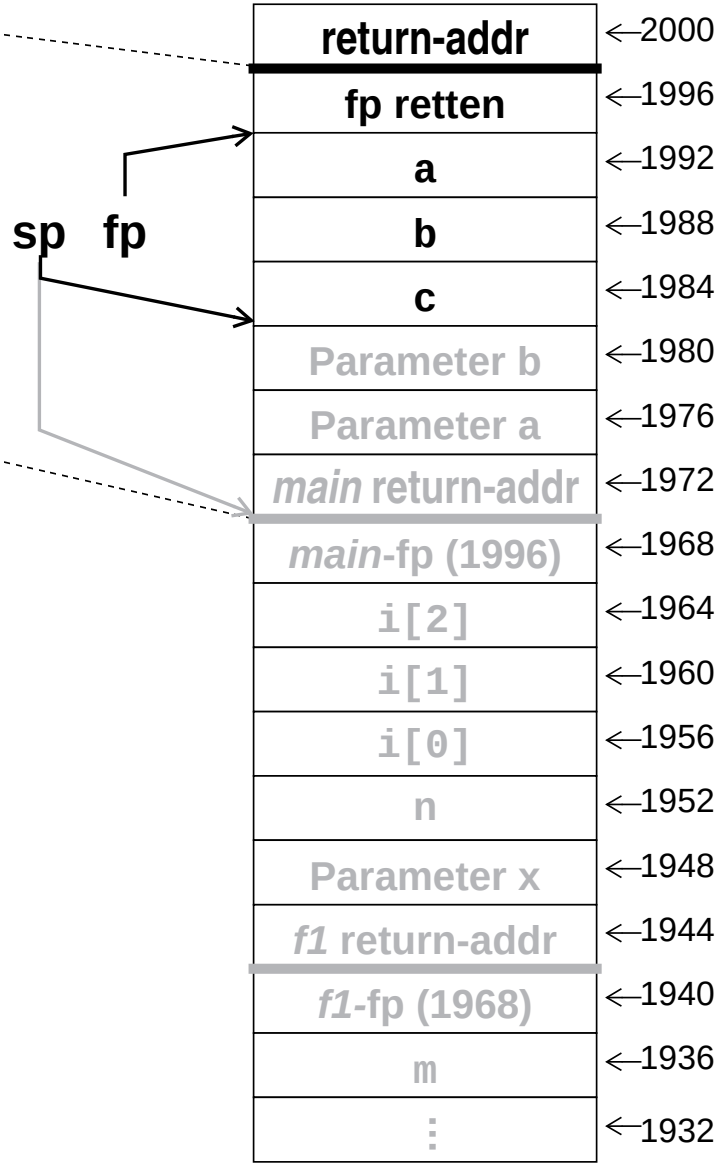
2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



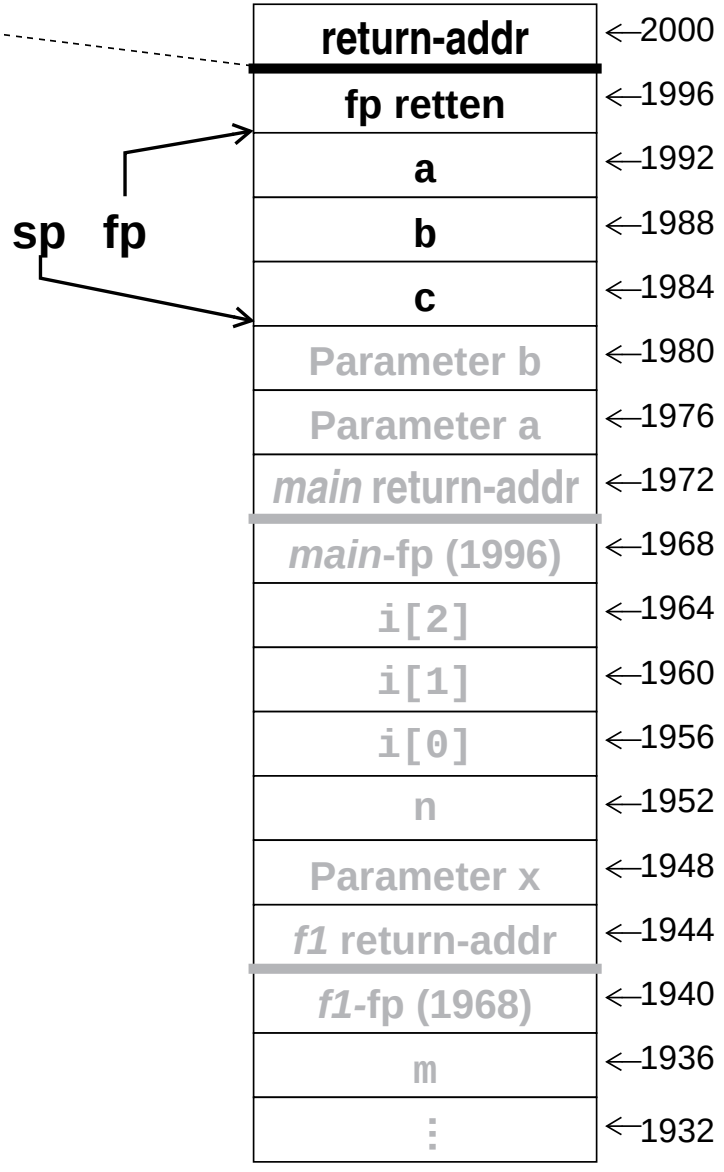
2 Stack mehrerer Funktionsaufrufe

```
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);
}
```



2 Stack mehrerer Funktionsaufrufe

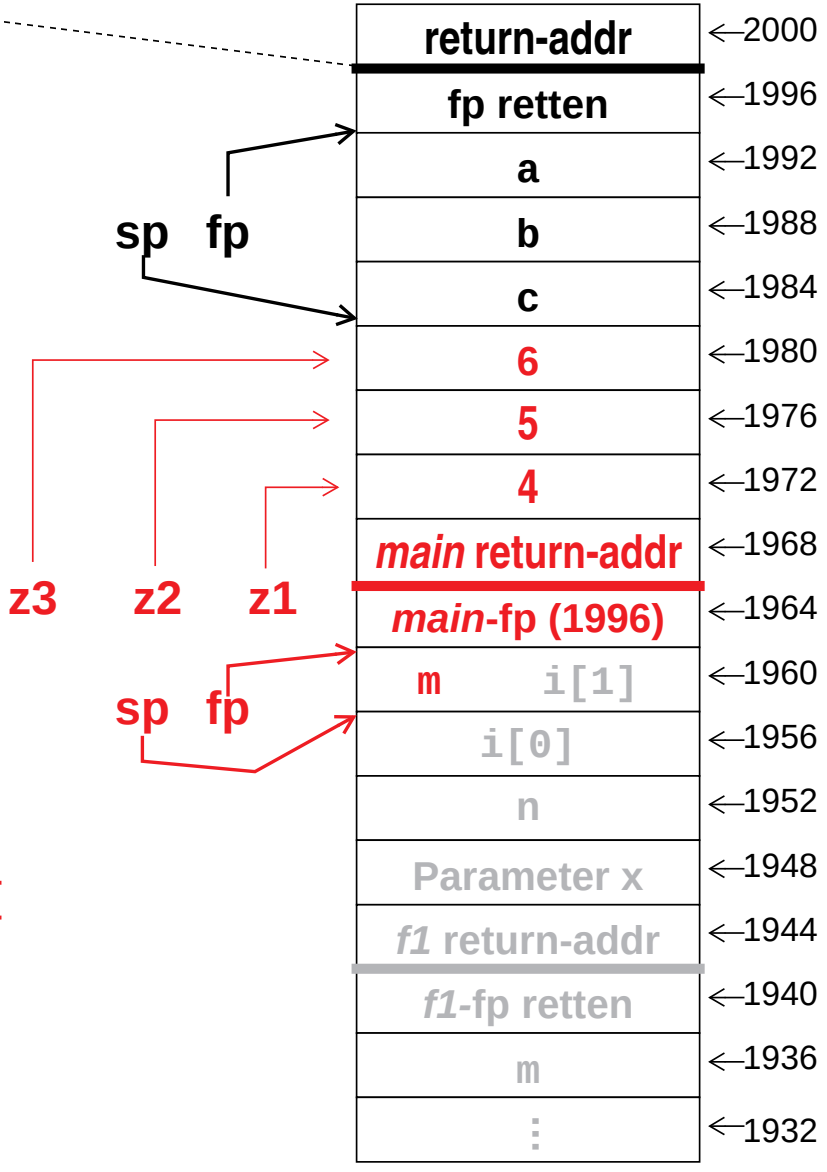
```
int main() {
        int a, b, c;

        a = 10;
        b = 20;

        f1(a, b);

        return(a);
}
```



2 Stack mehrerer Funktionsaufrufe

```
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);
}
```