Konzepte systemnaher Programmierung

Technische Hochschule Mittelhessen

Andre Rein

— Variablen —

Variablen in der Ninja VM

Bis jetzt kennen wir nur die Möglichkeiten, direkt mit dem Stack zu interagieren: `rdint`, `pushc`, `pop`. Auf Sprachebene gibt es bis jetzt also noch kein Konzept, wie Variablen in der Ninja VM abgebildet werden.

Als nächstes Thema führen wir nun **Variablen** ein und besprechen, welche Arten es gibt und wie diese in der **NinjaVM** behandelt werden:

- Globale Variablen
- Lokale Variablen
- Sichtbarkeit
- Gültigkeit

Vergleich Java / C

```java
class C {
    static int clsVar; 1
    int instVar; 2
    void m(int paramVar){ 3
        int localVar; 4
    }
}
```

1. Klassenvariable - Pro Klasse (nicht pro Objekt)
2. Instanzvariable - Pro Instanz (pro Objekt)
3. Parametervariable - Pro Aufruf der Methode
4. Lokale Variable - Pro Aufruf der Methode

Java		C
Klassenvariable	↔	Globale/Statische Variable
Instanzvariable	↔	Variable im Heap
Parametervariable	↔	Parametervariable
Lokale Variable	↔	Lokale Variable

Auch in Ninja benötigen wir diese Arten von Variablen, um vernünftig programmieren zu können.

Global vs. Statisch

- **Global**: beschreibt die Sichtbarkeit einer Variable
  - *Von wo* kann auf diese Variable zugegriffen werden
- **Statisch**: beschreibt die Lebensdauer der Variablen
  - *Wo* ist der Speicherplatz der Variablen (*wo lebt die Variable*)?
  - *Wann* hat die Variable Speicher → Über die komplette Ausführungszeit des Programms!
  - Vgl. Laufzeitspeicher Segment für statische Daten (static Data)
- Ein **globale** Variable ist per Definition auch immer **statisch** (global ⇒ statisch)
- Jedoch gibt es **statische** Variablen die **nicht global** sind (statisch ⇏ global)

Global vs. Statisch: Beispiel in C

```c
#include <stdio.h>

int f(void){
    static int i=0; 1
    i++;
    return i;
}

int main(int argc, char *argv[]){

    for (int i = 0; i < 8; ++i) {
        printf("loop [%d] f()=%d\n", i, f()); 2
    }
    return 0;

}
```

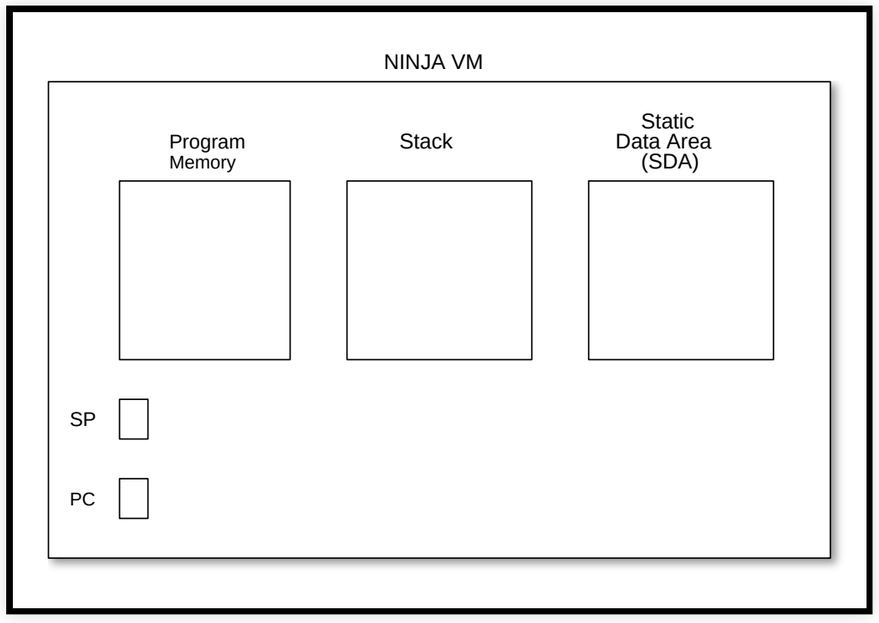
```
$ gcc -Wall -g  global_static.c -o global_static
$ ./global_static
loop [0] f()=1
loop [1] f()=2
loop [2] f()=3
loop [3] f()=4
loop [4] f()=5
loop [5] f()=6
loop [6] f()=7
loop [7] f()=8
```

1. Schlüsselwort `static` - Initialisierung `int i=0;` erfolgt **einmalig**!
2. Mehrfachaufruf von `f()`

- Ein Zugriff auf `i` kann nur aus der Funktion `f()` erfolgen. → Ist **keine globale** Variable
- Die Variable liegt jedoch auch im **statischen Datenbereich** und behält den Wert über die Funktionsaufrufe hinaus → **statisch**
- Die Variable **ist** also statisch, jedoch **nicht** global!

NinjaVM: Globale Variablen

Zur Speicherung und Verwaltung von globalen Variablen führen die sog. *Static Data Area* (**SDA**) ein. Die SDA kann als ein Array (*bestimmter Größe*) von **int**-Werten abgebildet wird.



- In der **SDA** werden alle **globalen Variablen** gespeichert und verwaltet
- Jede globale Variable, die angelegt wird, gibt es exakt **einmal** in der SDA
  - Unsere Variablen (auf NinjaVM Ebene) haben keinen Namen — Der Zugriff erfolgt einen Index

- Haben wir also z.B. eine SDA der Größe von 5 `int sda[5];` , gibt es Speicherplatz für 5 globale Variablen `sda[0], sda[1], … sda[4]` . Der Zugriff erfolgt mittels Indices

> Die Anzahl an globalen Variablen eines Ninja Programms sind dem Compiler bekannt und als Teil des Programm beim initialen Laden abrufbar. (Vgl. Ninja Binärformat)

NinjaVM: Globale Variablen

- `pushg <n>` → `... -> ... value` - Das n-te Element der SDA wird auf dem Stack abgelegt
- `popg <n>` → `... value -> ...` - Der Wert `value` wird in der SDA als n-tes Element gespeichert

Generell gilt: Ein `push` legt immer etwas auf den Stack und ein `pop` nimmt immer etwas vom Stack herunter.

Beispiel: `x = 3 * x + y`

Gegeben in SDA:

Name	Position in SDA
x	2
y	4

Ninja Assembler Code

```
pushc 3 // Lege Konstante 3 auf den Stack
pushg 2 // Lege x auf den Stack
mul     // Multiplikation: Ergebnis auf Stack
pushg 4 // Lege y auf den Stack
add     // Addition: Ergebnis auf Stack
popg 2  // Speichern des Wertes in x
```

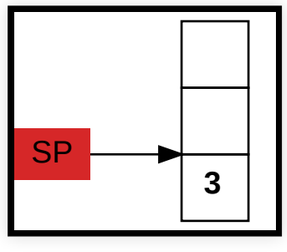
Ninja VM Globale Variablen: Beispiel

SDA

sda[4]	4	*y*
sda[3]		
sda[2]	3	*x*
sda[1]		
sda[0]		

```
ASSEMBLER
----------
pushc 3 <-- PC
pushg 2
mul
pushg 4
add
popg 2
```

Stack

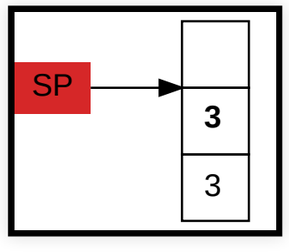


SDA

sda[4]	4	*y*
sda[3]		
sda[2]	3	*x*
sda[1]		
sda[0]		

```
ASSEMBLER
----------
pushc 3
pushg 2 <-- PC
mul
pushg 4
add
popg 2
```

Stack

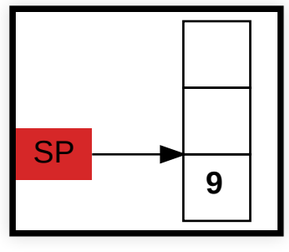


SDA

sda[4]	4	*y*
sda[3]		
sda[2]	3	*x*
sda[1]		
sda[0]		

```
ASSEMBLER
----------
pushc 3
pushg 2
mul     <-- PC
pushg 4
add
popg 2
```

Stack

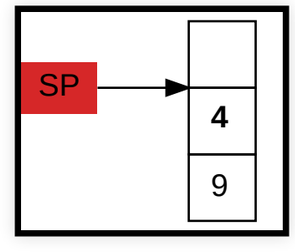


SDA

sda[4]	4	*y*
sda[3]		
sda[2]	3	*x*
sda[1]		
sda[0]		

```
ASSEMBLER
----------
pushc 3
pushg 2
mul
pushg 4 <-- PC
add
popg 2
```

Stack



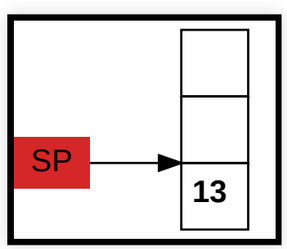
NinjaVM Globale Variablen: Beispiel

SDA

sda[4]	4	*y*
sda[3]		
sda[2]	3	*x*
sda[1]		
sda[0]		

```
ASSEMBLER
----------
pushc 3
pushg 2
mul
pushg 4
add     <-- PC
popg 2
```

Stack

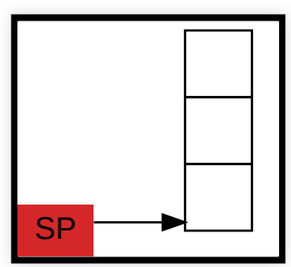


SDA

sda[4]	4	*y*
sda[3]		
sda[2]	**13**	*x*
sda[1]		
sda[0]		

```
ASSEMBLER
----------
pushc 3
pushg 2
mul
pushg 4
add
popg 2  <-- PC
```

Stack



C: Lokale Variablen

- Lokale Variablen leben innerhalb eines Funktionsaufrufs, d.h. jeder Aufruf der Funktion `f()` hat eine eigene Variable `i`, die nur innerhalb dieses Funktionsaufrufs zur Verfügung steht

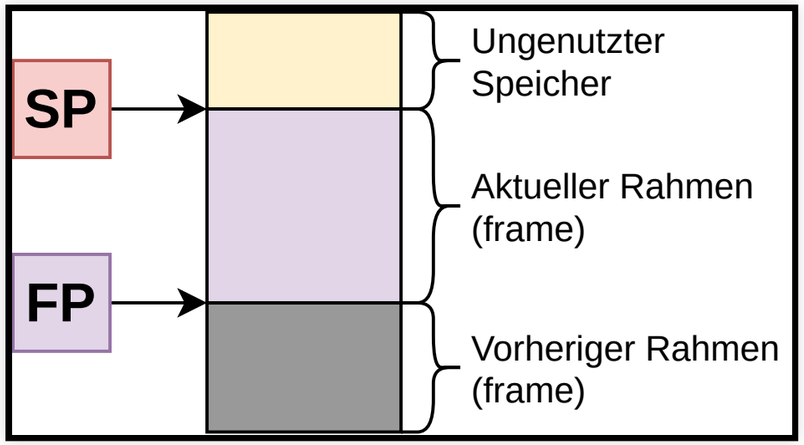
c -Code

```c
int f(void){
    int i = 10;
    return i;
}
```


Lokale Variablen werden in `C` auf dem Stack verwaltet. So erhält jede Funktion beim Aufruf eigene lokale Variablen. Technisch ist dies als sog. Rahmen (engl. frame) umgesetzt, auf X86_64 z.B. aus dem Zusammenspiel der Register `RSP` (Stackpointer) und einem `RBP`, dem sog. Framepointers (FP).

NinjaVM: Rahmen/Frames

Der Bereich zwischen **Framepointer** (FP) und **Stackpointer** (SP) wird *Rahmen* (**Frame**) genannt. Der Frame beginnt an der $Position$ auf dem Stack, auf die der FP aktuell zeigt und endet an $Position - 1$, auf die der SP aktuell zeigt.



- Beispiel:
  - `FP=7, SP=12` → Frame: `7 - 11`. Größe `20 Byte` (`int`)
  - `FP=17, SP=37` → Frame: `17 - 36`. Größe `80 Byte` (`int`)

NinjaVM: Lokale Variablen

Durch die Einführung des **Frame**-Konzeptes ist es möglich, einen einzelnen Funktionsaufruf auf dem Stack zu verwalten (d.h. also zu *identifizieren*).

Dies erlaubt uns u.a., dass wir Speicherplatz für lokale Variablen in einem gesonderten Bereich des Frames (direkt oberhalb von FP) *reservieren* und letztendlich dort auch Variablen *speichern* können.

- **Instruktionen** zur Verwaltung der Frames:
  - `asf <n>` *Allocate Stack Frame* — `n` gibt die Anzahl der zu reservierenden lokalen Variablen an
  - `rsf` *Release Stack Frame* — Entfernen des aktuellen Stackframes und Rückkehr zum vorherigen Stackframe

NinjaVM: Lokale Variablen

Die beiden Funktionen `asf n` und `rsf` modifizieren die Register und repositionieren den Stack. Aufgrund der Repositionierung macht es also keinen Sinn, die Effekte auf den Stack zu betrachten (`vorher ... nachher`).

Daher wird stattdessen die **Semantik** der Instruktionen beschrieben.

`asf <n>`

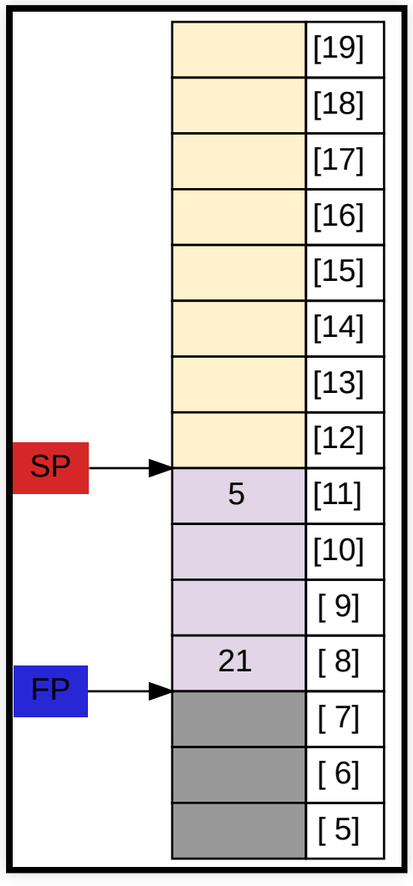
```
push(fp)  // sichert aktuellen FP auf dem Stack
fp = sp   // setzt den neuen Beginn des Rahmens
sp = sp+n // reserviert n-Anzahl an Variablen im Frame
```

`rsf`

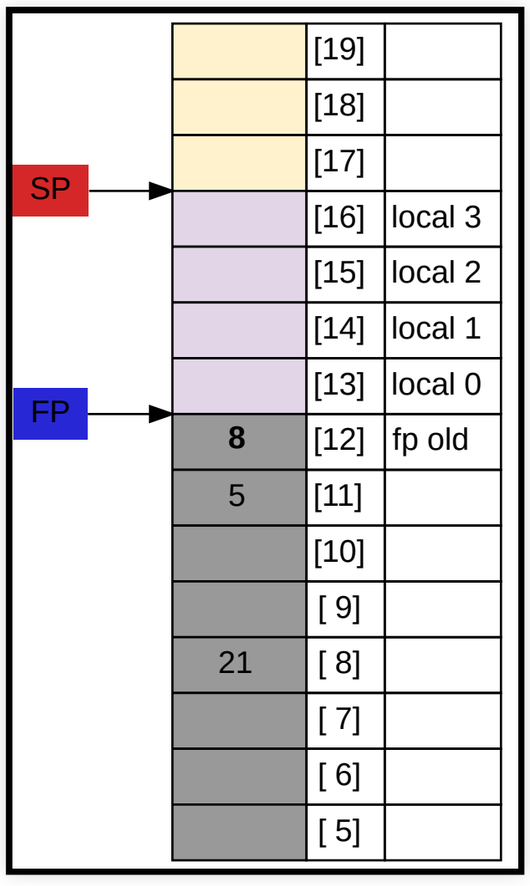
```
sp = fp    // zeigt auf alten FP Wert
fp = pop() // setzt FP auf alten Wert
```

NinjaVM: Lokale Variablen

Stack Ausgangslage

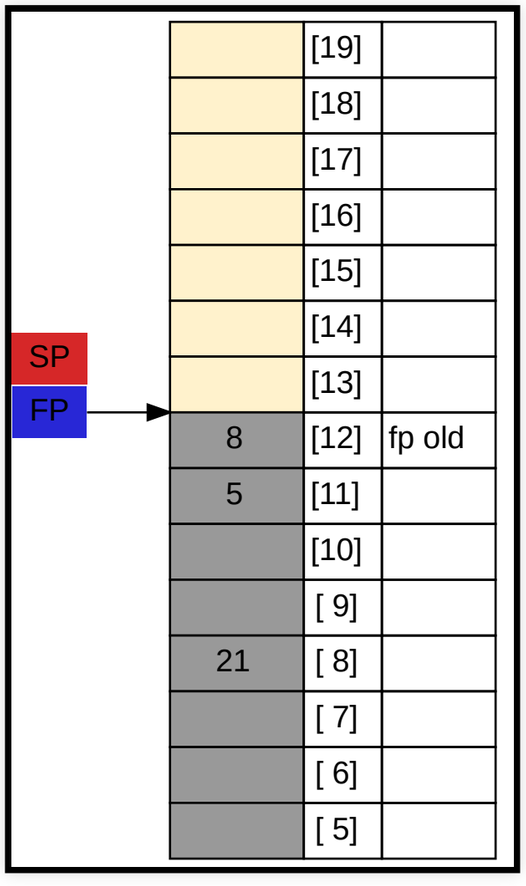


Stack `asf 4`



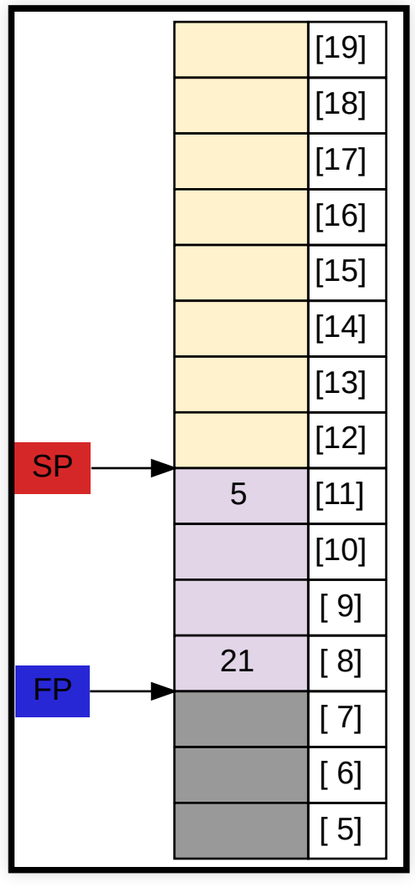

```
push(fp)  // push(8)
fp = sp   // fp = 13
sp = sp+4 // sp = 17
```

Stack rsf Schritt 1




```
sp = fp // sp = 13 <-
fp = pop()
```

Stack rsf

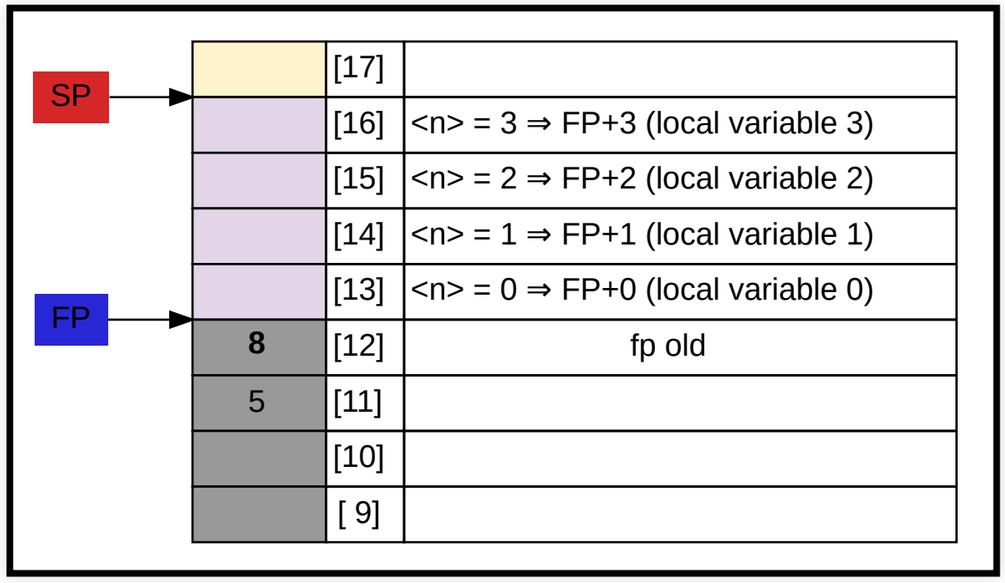



```
sp = fp
fp = pop() // fp = 8 <-
```

NinjaVM: Lokale Variablen

Die beiden Instruktionen, um lokale Variablen im Stackframe zu verwalten, sind `pushl` und `popl`. Diese beiden Instruktionen sind funktional sehr ähnlich zu den Instruktionen zur Verwaltung der globalen Variablen. In diesem Fall ist jedoch der Speicherplatz auf dem **Stack** und nicht in der SDA.

- `pushl <n> ... -> ... value`
  - *Die n-te lokale Variable wird auf den Stack gelegt.*
- `popl <n> ... value -> ...`
  - *Die oberste Variable wird als n-te lokale Variable gespeichert.*



Die n-te lokale Variable liegt hierbei genau n-Positionen oberhalb des Framepointers FP. Sie wird also **relativ** zur Position von FP angegeben! Der Immediate Wert bestimmt hierbei die Position.

NinjaVM: Lokale Variablen Beispiel

Gegeben:

Name	n	Stackindex
x	2	FP+2
y	4	FP+4

Ninja Assembler Code

```
pushc 3 // Lege Konstante 3 auf Stack
pushl 2 // Lege x (Position FP+2) auf den Stack
mul     // Multiplikation: Lege Ergebnis auf Stack
pushl 4 // Lege y (Position FP+4) auf den Stack
add     // Addition: Lege Ergebnis auf Stack
popl 2  // Speichern des Wertes in x (Position FP+2)
```

Die Ausführung der Instruktionen zur Berechnung des Ausdrucks ist weitestgehend äquivalent zum gezeigten Ablauf, der bei den globalen Variablen verwendet wurde. Der Unterschied ist jedoch: Der Speicherplatz der **lokalen Variablen** ist nun wie besprochen im **Stackframe** (und nicht in der SDA)!

Instruktionen: Globale und Lokale Variablen

Instruktion	Opcode	Stack Layout
`pushg <n>`	`11`	`... -> ... value`
`popg <n>`	`12`	`... value -> ...`
`asf <n>`	`13`	
`rsf`	`14`	
`pushl <n>`	`15`	`... -> ... value`
`popl <n>`	`16`	`... value -> ...`

Übersicht NinjaVM

Abschließend nun die NinjaVM in der Übersicht:

- mit Static Data Area (**SDA**) zur Speicherung **globaler Variablen** und
- mit Framepointer (**FP**) zur Verwaltung von **Frames** (Rahmen), die zur Speicherung **lokaler Variablen** verwendet werden