Konzepte systemnaher Programmierung

Technische Hochschule Mittelhessen

Andre Rein

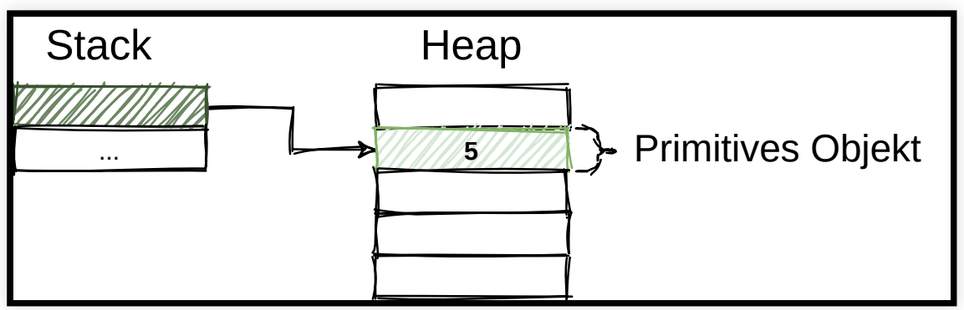
— Ninja Objekte —

Speicherung von Objekten in Ninja

Die Speicherung von Ninja-Rechenobjekten erfolgt bis jetzt in Form sog. primitiver Objekte auf dem Heap.

Ehemals wurden im Payload der primitiven Objekte Integerwerte verwaltet. *Seit Einführung der BigInt-Bibliothek sind dort nun Objekte vom Typ `Big` gespeichert.*

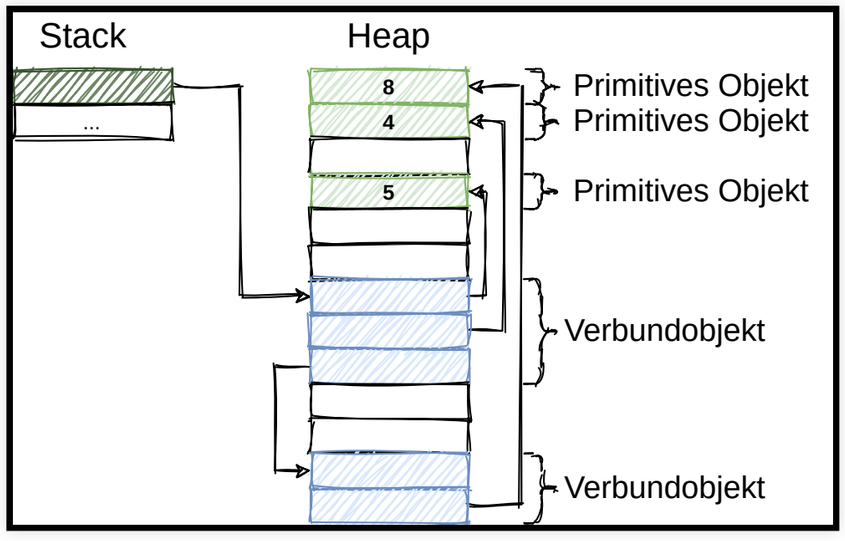
Zusätzlich wird für jedes Rechenobjekt eine Datenstruktur auf dem Stack verwaltet, die eine Referenz auf das zugehörige Rechenobjekt im Heap speichert.



Auf dem Stack existieren auch noch *normale* Zahlen (für *Rücksprungadresse* und *Framepointer*), die ohne Hilfe von Objektreferenzen verwaltet werden.

Verbundobjekte in Ninja

In Ninja können Objekte andere Ninja Objekte *beinhalten*, indem sie Variablen zur Speicherung von Objektreferenzen bereitstellen. Diese Art von Objekten sind sog. **Verbundobjekte** oder **zusammengesetzte Objekte** (englisch **composite object**).



- Verbundobjekte stellen **Speicher** bereit, um Objektreferenzen (*Speicheradressen*) auf *andere Objekte* zu speichern — *Sie agieren somit als eine Art Container, um andere Objekte zu verwalten*
- *Andere Objekte* können **primitive Objekte** oder wiederum **Verbundobjekte** sein

Grundsätzliche Darstellung von Verbundobjekten

Die Verwaltung von Objekten erfolgt in der VM über die Datenstruktur `ObjRef`. Da beliebige Daten im *Payload* von `ObjRef` verwenden werden können, können somit auch Referenzen auf andere Objekte (*d.h. Speicheradressen von anderen Objekten*) dort gespeichert und verwaltet werden.

```
typedef struct {
    unsigned int size;     // # byte of payload
    unsigned char data[1]; // payload
} *ObjRef;
```

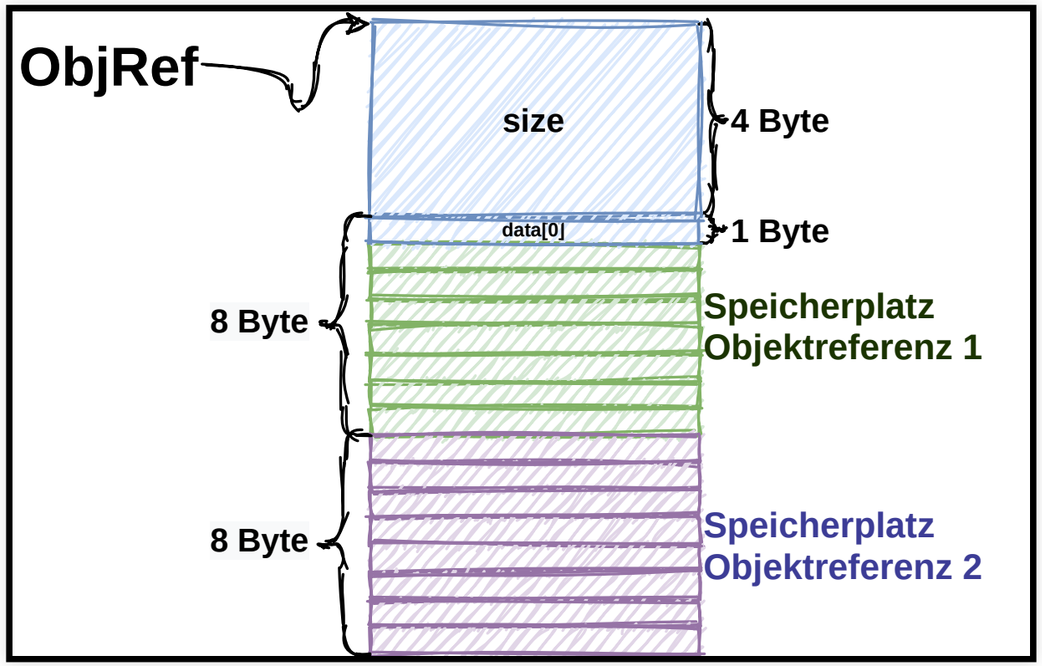
Die Datenstruktur `ObjRef` kann also für *primitive* **und** *zusammengesetzte Objekte* gleichermaßen verwendet werden.

Die enthaltenen Daten werden nur anders interpretiert!

Beispiel: ObjRef mit Objektreferenzen

Anmerkung: *Speicheradressen auf X86_64 benötigen `8 Byte` Speicherplatz (`sizeof (void *)`).*
Um ein `ObjRef` mit 2 Speicherplätzen zu erzeugen, in denen jeweils eine Referenz auf ein weiteres Objekt gespeichert werden soll, müssen `2 * 8 = 16` Bytes für den **Payload** reserviert werden.

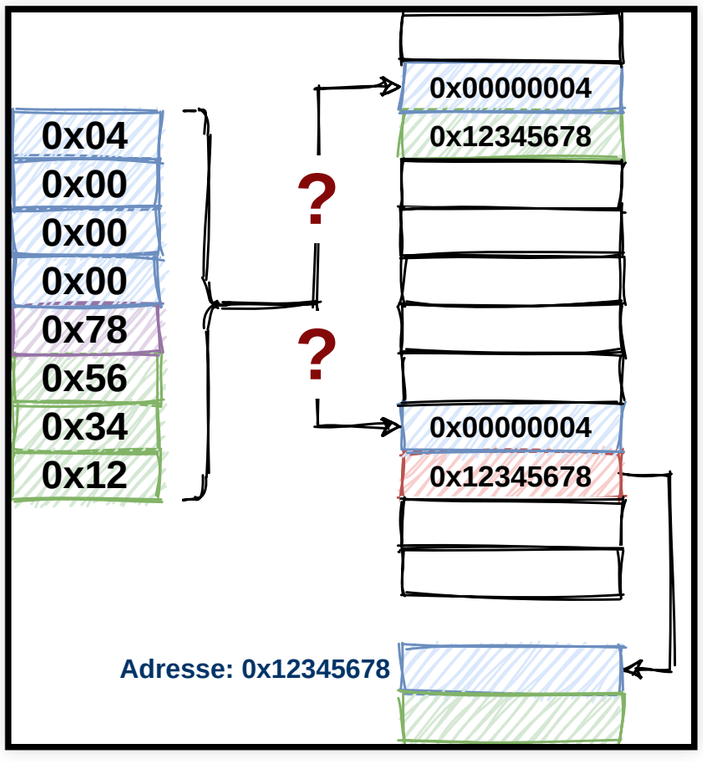
```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    unsigned int size;      // # byte of payload
    unsigned char data[1]; // payload data, size as needed!
} *ObjRef;

int main(int argc, char *argv[]) {
    ObjRef cmpObj;
    unsigned int objSize;
    objSize = sizeof(*cmpObj) + (2 * sizeof(void *));
    if ((cmpObj = malloc(objSize)) == NULL) {
        perror("malloc");
    }
    /* simulate 2 arbitrary pointer addresses*/
    ((ObjRef *)cmpObj->data)[0]=malloc(8);
    ((ObjRef *)cmpObj->data)[1]=malloc(8);

    cmpObj->size=2; /* count of pointer elements */
    printf("sizeof(cmpObj) =%lu\n", objSize);
    printf("cmpObj elements=%lu\n", cmpObj->size);
    printf("cmpObj->data[0]=%p\n", ((ObjRef *) cmpObj->data)[0]);
    printf("cmpObj->data[1]=%p\n", ((ObjRef *) cmpObj->data)[1]);
    free(cmpObj);
    return 0;
}
```



Unterscheidung zwischen Objekttypen

Da die Datenstruktur `ObjRef` nun entweder ein **primitives Objekt** oder eine **Verbundobjekt** sein kann, muss man zwischen beiden Typen unterscheiden können.

- Ohne eine explizites Unterscheidungsmerkmal kann man nicht entscheiden:
  - Ob enthaltene Daten im Payload eine Speicheradresse oder
  - andere Daten (z.B. `int` oder `Big` *vgl. BigInt Bibliothek*) repräsentieren.



Zum besseren Verständnis wurde im obigen Beispiel der Datentyp `int` verwendet.

Unterscheidung zwischen Objekttypen

Zur Unterscheidung gibt es u.a. zwei Möglichkeiten:

1. Speichern der Typinformation im höchstwertigen Bit (MSB) von `size`
2. Einführen einer zusätzlichen Komponente in `ObjRef` — z.B. `isCmpObj`

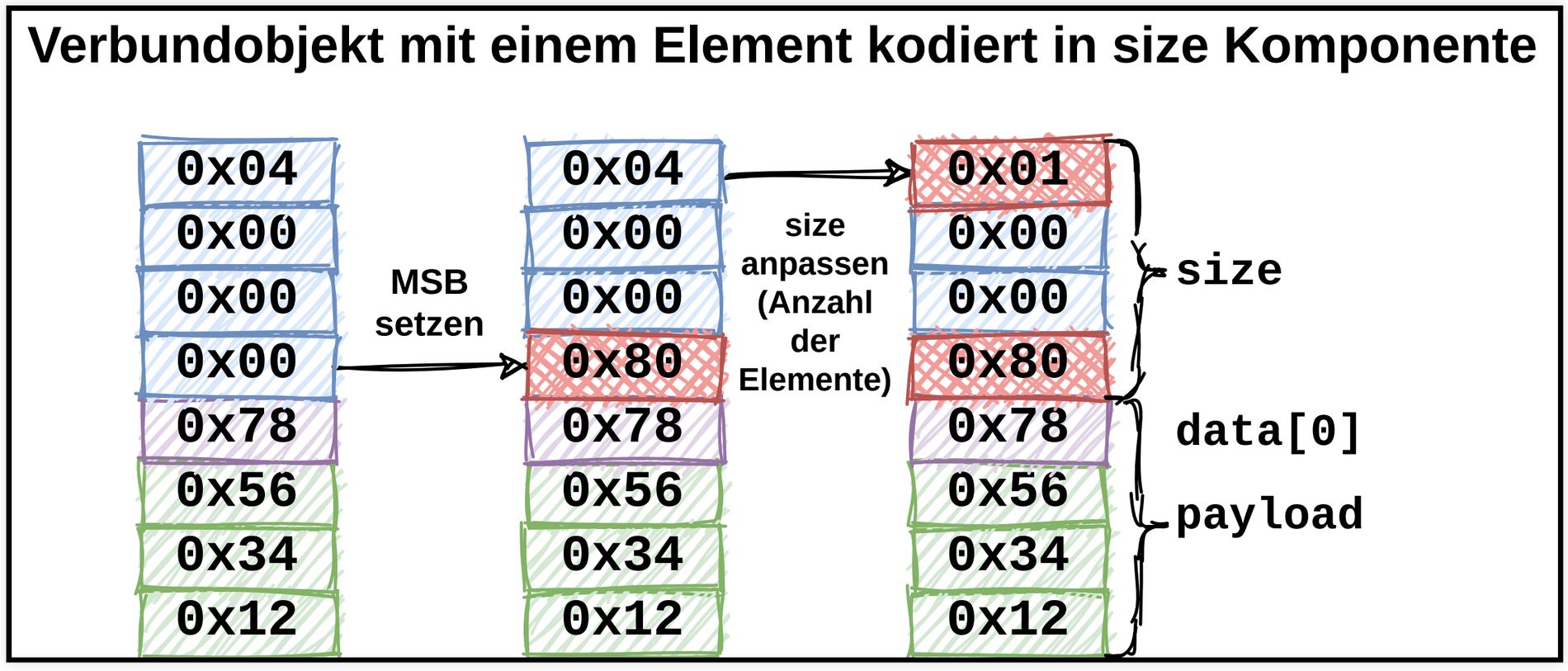
Unterscheidung Objekttypen: Kodierung in `size`

Die `size`-Komponente von `ObjRef` kann dazu verwendet werden, um eine Unterscheidung der Objekttypen zu realisieren.

- Um ein Objekt als Verbundobjekt zu klassifizieren wird:
  - Das höchstwertige Bit von `size` auf `1` gesetzt
- Im Payload von Verbundobjekten sind *ausschließlich* Objektreferenzen (d.h. Zeiger) gespeichert — in der Komponente `size` wird daher nun die **Anzahl** der enthaltenen Objektreferenzen gespeichert
  - Und **nicht mehr** die Anzahl an belegten Bytes

An `ObjRef` für primitive Objekte findet keine Veränderung statt, sie bleiben genau so erhalten wie bisher!

Unterscheidung Objekttypen: Kodierung in `size`



Unterscheidung Objekttypen: Makros

- Generierung einer Bitmaske für MSB
  - `#define MSB (1 << (8 * sizeof(unsigned int) - 1))`
- Handelt es sich um ein primitives Objekt?
  - `#define IS_PRIMITIVE(objRef) (((objRef)->size & MSB) == 0)`
- Wie viele Objektreferenzen enthält das Objekt?
  - `#define GET_ELEMENT_COUNT(objRef) ((objRef)->size & ~MSB)`

Unterscheidung Objekttypen: Makros

Generierung einer Bitmaske für MSB

```
#define MSB (1 << (8 * sizeof(unsigned int) - 1))
#define MSB (1 << (8 * 4 - 1))
#define MSB (1 << (32 - 1))
```

- Semantisch äquivalent zu `#define MSB (1 << 31)`
  - **Binär:** `0b1000 0000 0000 0000 0000 0000 0000 0000`
  - **Hexadezimal:** `0x80000000`

Unterscheidung Objekttypen: Makros

Handelt es sich um ein primitives Objekt?

```
#define IS_PRIMITIVE(objRef) (((objRef)->size & MSB) == 0)
```

Beispiel: Verbundobjekt mit 7 Elementen `objRef->size=0x80000007`

`((objRef)->size & MSB)` → `2147483648`

```
  1000 0000 0000 0000 0000 0000 0000 0111 | objRef->size
& 1000 0000 0000 0000 0000 0000 0000 0000 | MSB
  ------------------------------------- |
  1000 0000 0000 0000 0000 0000 0000 0000 | 2147483648
```

`(2147483648 == 0)` → `0` (also `False`)

`#define IS_PRIMITIVE(objRef)` → `0` (also `False`) → **Verbundobjekt**

Unterscheidung Objekttypen: Makros

Wie viele Objektreferenzen enthält das Objekt?

```
#define GET_ELEMENT_COUNT(objRef) ((objRef)->size & ~MSB)
```

Beispiel: Verbundobjekt mit 7 Elementen `objRef->size=0x80000007`

`((objRef)->size & ~MSB)` →

`7`

```
  1000 0000 0000 0000 0000 0000 0000 0111 | objRef->size (0x80000007)
& 0111 1111 1111 1111 1111 1111 1111 1111 | ~MSB (0x7FFFFFFF)
  ------------------------------------- |
  0000 0000 0000 0000 0000 0000 0000 0111 | 7
```

`#define GET_ELEMENT_COUNT(objRef)` → `7` → Enthält 7 **andere Elemente**

Zugriff auf Objektreferenzen: Makros

Um auf Objekte innerhalb von Verbundobjekten zugreifen zu können, müssen die Daten im Payload als Adressen zu einem `ObjRef` interpretiert und umgewandelt (*gecastet*) werden. Dies ist notwendig, damit der Compiler weiß, auf welche Daten er wie zugreifen kann.

Der Zugriff auf Objektreferenzen innerhalb von Verbundobjekten kann mit folgendem Makro vereinfacht werden:

```
#define GET_REFS_PTR(objRef) ((ObjRef *) (objRef)->data)
```

Hierbei werden die in `objRef→data` enthaltenen Daten als Zeiger auf ein Objekt vom Typ `ObjRef` interpretiert, d.h. als eine Speicheradresse mit `8` -Byte.

Zugriff auf Objektreferenzen: Makros

Makroverwendung

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


#define GET_REFS_PTR(objRef) ((ObjRef *) (objRef)->data)

typedef struct {
    unsigned int size;     // # byte of payload
    unsigned char data[1]; // payload data, size as needed!
} *ObjRef;


int main(int argc, char *argv[]) {
    ObjRef cmpObj;
    unsigned int objSize;
    objSize = sizeof(*cmpObj) + (2 * sizeof(void *));
    if ((cmpObj = malloc(objSize)) == NULL) {
        perror("malloc");
    }
    cmpObj->size=80000002; /* count of pointer elements */

    /* simulate 2 arbitrary pointer addresses*/
    GET_REFS_PTR(cmpObj)[0]=malloc(8);
    GET_REFS_PTR(cmpObj)[1]=malloc(8);

    /* create primObj with 1 integer in payload */
    GET_REFS_PTR(cmpObj)[0]->size=4;

    /* set the payload to an 5 as integer*/
    *(int *) GET_REFS_PTR(cmpObj)[0]->data=5;

    /* create primObj with 1 integer in payload */
    GET_REFS_PTR(cmpObj)[1]->size=4;

    /* set the payload to an 7 as integer*/
    *(int *) GET_REFS_PTR(cmpObj)[1]->data=7;


    printf("cmpObj->data[0]=%p\n", GET_REFS_PTR(cmpObj)[0]);
    printf("cmpObj->data[1]=%p\n", GET_REFS_PTR(cmpObj)[1]);
    printf("cmpObj->data[0]->data [%p] = 0x%08x\n",
            GET_REFS_PTR(cmpObj)[0],
            *(int *)GET_REFS_PTR(cmpObj)[0]->data);
    printf("cmpObj->data[1]->data [%p] = 0x%08x\n",
            GET_REFS_PTR(cmpObj)[1],
            *(int *)GET_REFS_PTR(cmpObj)[1]->data);
    return 0;
}
```

Ohne Makroverwendung

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


#define GET_REFS_PTR(objRef) ((ObjRef *) (objRef)->data)

typedef struct {
    unsigned int size;     // # byte of payload
    unsigned char data[1]; // payload data, size as needed!
} *ObjRef;


int main(int argc, char *argv[]) {
    ObjRef cmpObj;
    unsigned int objSize;
    objSize = sizeof(*cmpObj) + (2 * sizeof(void *));
    if ((cmpObj = malloc(objSize)) == NULL) {
        perror("malloc");
    }
    cmpObj->size=80000002; /* count of pointer elements */

    /* simulate 2 arbitrary pointer addresses*/
    ((ObjRef *)cmpObj->data)[0]=malloc(8);
    ((ObjRef *)cmpObj->data)[1]=malloc(8);

    /* create primObj with 1 integer in payload */
    ((ObjRef *)cmpObj->data)[0]->size=4;

    /* set the payload to an 5 as integer*/
    *(int *) ((ObjRef *)cmpObj->data)[0]->data=5;

    /* create primObj with 1 integer in payload */
    ((ObjRef *) cmpObj->data)[1]->size=4;

    /* set the payload to an 7 as integer*/
    *(int *) ((ObjRef *)cmpObj->data)[1]->data=7;


    printf("cmpObj->data[0]=%p\n", ((ObjRef *)cmpObj->data)[0]);
    printf("cmpObj->data[1]=%p\n", ((ObjRef *)cmpObj->data)[1]);
    printf("cmpObj->data[0]->data [%p] = 0x%08x\n",
            ((ObjRef *)cmpObj->data)[0],
            *(int *)((ObjRef *)cmpObj->data)[0]->data);
    printf("cmpObj->data[1]->data [%p] = 0x%08x\n",
            ((ObjRef *)cmpObj->data)[1],
            *(int *)((ObjRef *)cmpObj->data)[1]->data);
    return 0;
}
```

Unterscheidung Objekttypen: ObjRef Komponente

Es gibt beliebig viele Varianten, wie man eine zusätzliche Komponente in `ObjRef` verwenden kann um eine Unterscheidung der Datentypen zu realisieren.

```
typedef struct {
    unsigned int size;
    unsigned char type; (1)
    unsigned char data[1];
} *ObjRef;
```

(1) Variante mit explizitem Typ, z.B. `'c'` → `composite` und `'p'` → `primitive`

```
typedef struct {
    unsigned int size;
    bool isCmpObj; (2)
    unsigned char data[1];
} *ObjRef;
```

(3) Variante mit `bool` aus `stdbool.h`

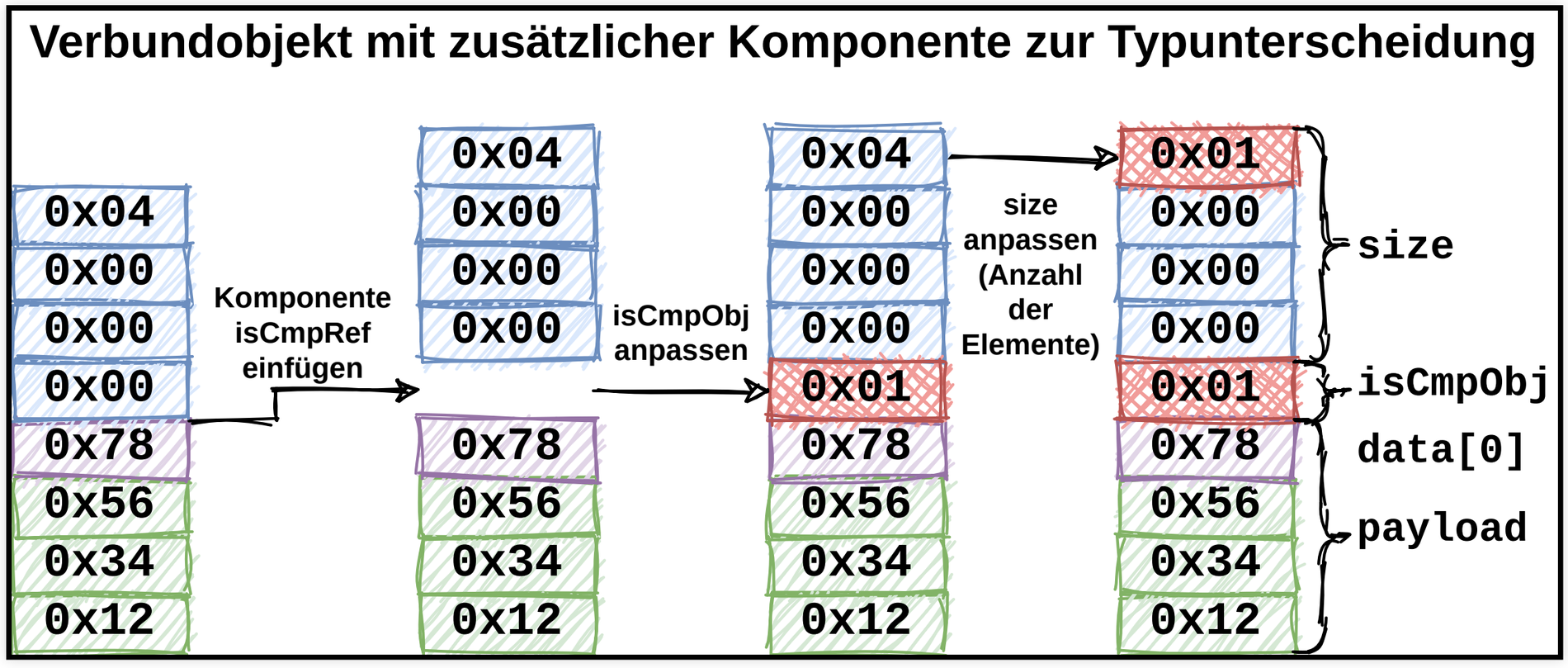
```
typedef struct {
    unsigned int size;
    unsigned char isPrimObj; (3)
    unsigned char data[1];
} *ObjRef;
```

(3) Variante mit eigener Interpretation von `isPrimObj`

- **Vorteil** → Sehr gute Lesbarkeit und Verständnis der Lösung und optimierte Zugriffe auf die Komponenten
- **Nachteil** → Man benötigt mehr Speicher für jedes `ObjRef`

Die jeweiligen Lösungen, wie nun genau dieses Komponenten zur Unterscheidung verwendet werden, sind Implementierungsspezifisch.

Unterscheidung Objekttypen: ObjRef Komponente



Verbundobjekttypen

Auf der Sprachebene von **Ninja** gibt es zwei unterschiedliche Typen von Verbundobjekten:

1. **Records** — Zugriff auf Variable über Namen
   - In Ninja-Programmen greift man auf Werte in einem **Record** mittels einem Namen zu (Bsp. *Recordname* `foo` — *Variable* `x` → `foo.x`)
2. **Arrays** — Zugriff auf Variable über Index
   - In Ninja-Programmen greift man auf Werte in einem **Array** mittels einem Index zu (Bsp. *Arrayname* `bar` — *Variable an Speicherplatz* `5` → `bar[5]`)

Beides sind Konzepte aus der Programmiersprache Ninja. Wir müssen die unterschiedlichen Konzepte für Verbundtypen nun in die VM übertragen und diese unterstützen!

Verbundobjekttyp: Record (Ninja)

Definition eines Records:

```
type Point = record {
    Integer x;
    Integer y;
};
```

Die Datentypen können auch unterschiedlich sein. Die Definition eines Records ist auf VM Ebene nicht relevant!

Erzeugung und Zugriff von Variablen vom Type Point:

```
// create a point
local Point p;
p = new(Point)

// use a point
k=p.x;
p.y=2*k;
```

Verbundobjekttyp: Record (VM)

Ein neuer **Record** wird mit der Instruktion `new <n>` erzeugt, der Wert `<n>` gibt hierbei an, wie viele *andere Elemente* im Record verwaltet werden.

```
new <n> ... -> ... object
```

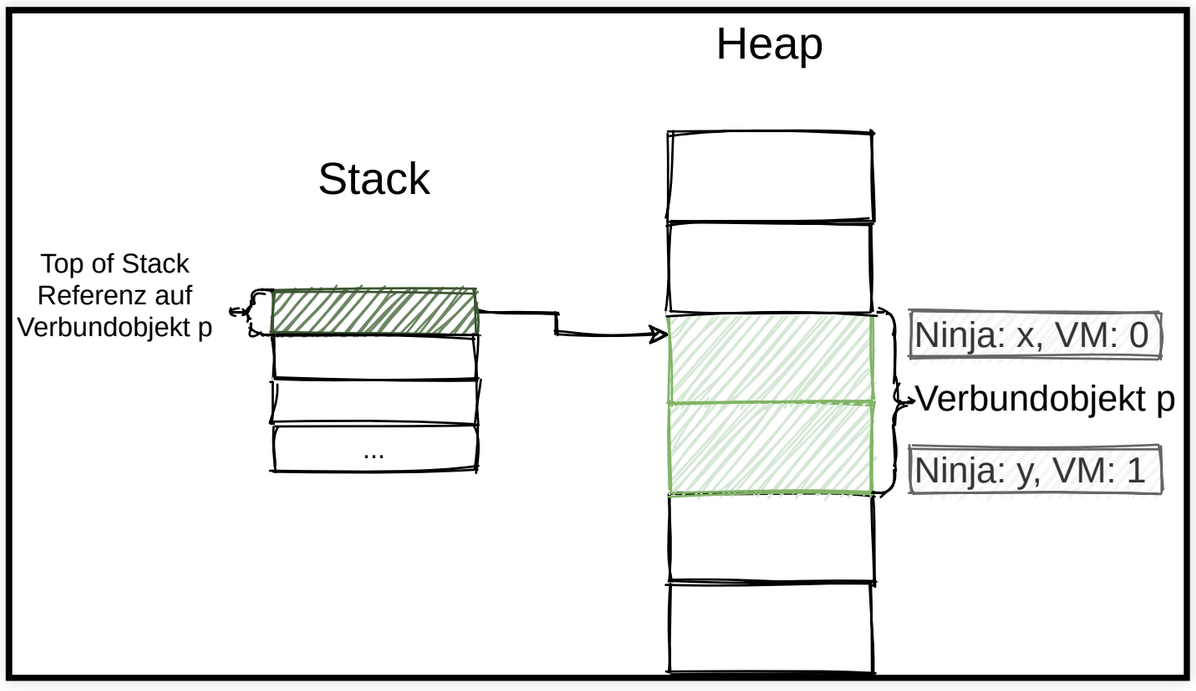
- Der Zugriff auf ein Verbundobjekt vom Typ Record erfolgt mittels `(...).component_name` (z.B. p.x)

Der Ninja-Compiler weiß, welche Komponente gemeint ist, wenn der Name der Komponente verwendet wird. Die Namen werden vom Compiler deshalb in Indices umgewandelt und der Zugriff erfolgt somit auf VM Ebene mit Indexwerten $(0, 1, 2, \ldots, n)$.

Verbundobjekttyp: Record (VM)

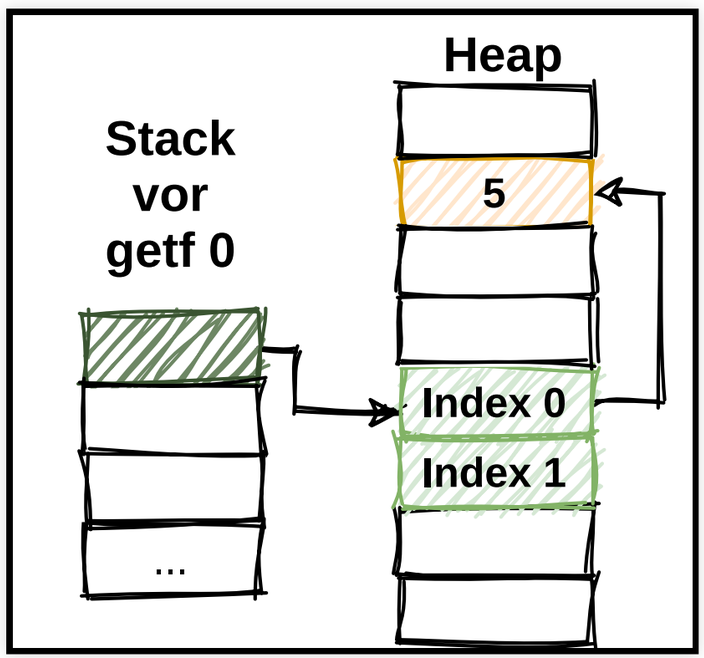
Annahme: Wir verwenden den Typ `Punkt p` mit 2 Integerwerten `p.x` und `p.y`

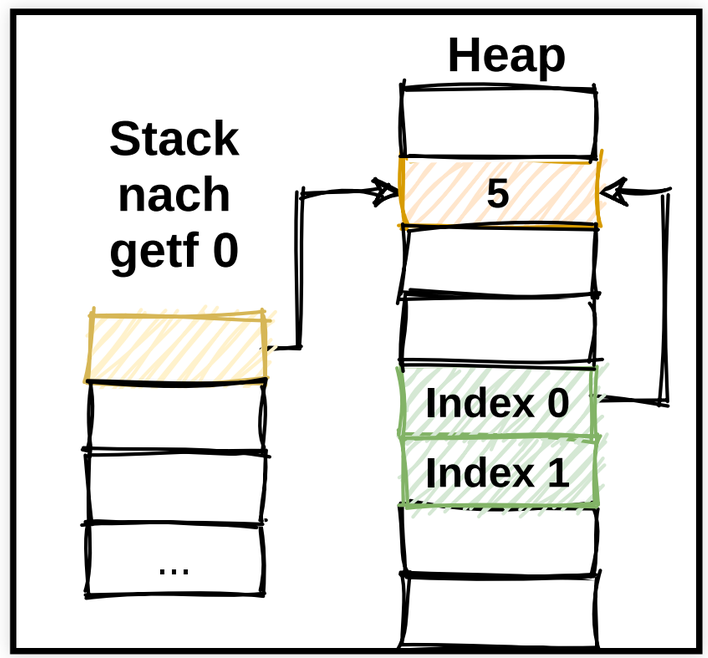
`new 2` Erzeugt ein Verbundobjekt mit 2 Speicherplätzen



Verbundobjekttyp: Record (VM)

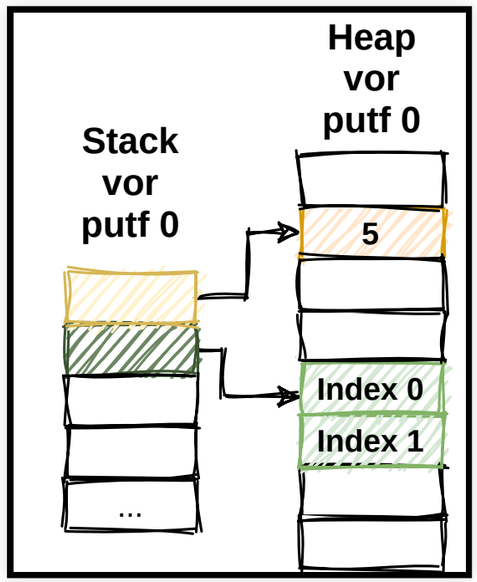
- **getfield**: `getf <i> ... object -> ... value` — Die im Verbundobjekt `object` an Position `i` enthaltene Referenz auf ein Objekt `value` wird auf dem Stack abgelegt

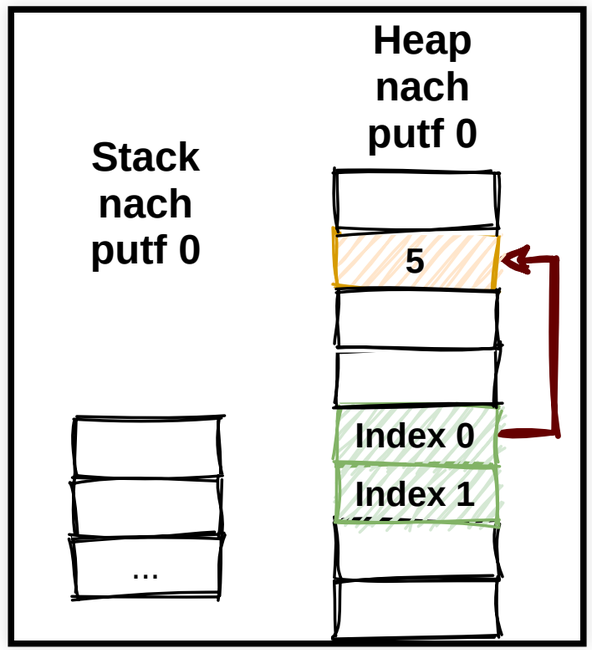




Verbundobjekttyp: Record (VM)

- **putfield**: `putf <i> ... object value -> ...` — Die Referenz auf ein Objekt `value` wird im Verbundobjekt `object` an Position `i` gespeichert. *Beide Operanden sind anschließend nicht mehr auf dem Stack*





Verbundobjekttyp: Arrays (Ninja)

Definition eines Arrays: `type Vector = Integer[];`

Erzeugung und Zugriff auf eine Variable vom Typ `Vector` (entspricht einem Array von Integer):

```
// create a vector
local Vector v;
// no. of Elements only known during runtime!
v=new(Integer[2*n+1]);

// use a vector
k=v[i]
v[n-i]=2*k
```

Verbundobjekttyp: Array (VM)

Ein neues **Array** wird mit der Instruktion `newa` erzeugt.

```
newa ... nelem -> ... array
```

`nelem` ist ein berechneter Wert, der angibt wie Groß das Array sein soll. Dies ist in diesem Fall kein Immediate-Wert.

Beispiel: Berechnung von `nelem` für `v=new(Integer[2*n+1]);`

Annahme: Der Wert `n` ist Zur Laufzeit bekannt und als globale Variable gespeichert!

```
pushc 2 // 2
pushg 1 // n
mul     // 2*n
pushc 1 // 1
add     // 2*n+1
```

Der Zugriff auf ein Verbundobjekt vom Typ Array erfolgt mittels `(...).[...]` (z.B. v[2])

Verbundobjekttyp: Arrays (VM)

Der Hauptunterschied zwischen **Records** und **Arrays** auf VM Ebene ist, dass bei Arrays die Anzahl der Elemente erst zur Laufzeit bekannt sind. Somit kann kein Code erzeugt werden, der Zugriffe mittels fest kodierter Indices verwendet. Ein ähnliches Verhalten gab es bereits bei **globalen** und **lokalen** Variablen. Der Zugriff auf Records ähnelt dem Zugriff auf globale Variablen und der Zugriff auf Arrays, dem Zugriff auf lokale Variablen.

Verbundobjekttyp: Array (VM)

- **getfield of array**: `getfa ... array index -> ... value` — Die im Verbundobjekt `array` an Position `index` enthaltene Referenz auf ein Objekt `value` wird auf dem Stack abgelegt
- **putfield of array**: `putfa ... array index value -> ...` — Die Referenz auf ein Objekt `value` wird im Verbundobjekt `array` an Position `index` gespeichert. *Alle Operanden sind anschließend nicht mehr auf dem Stack*

Der einzige signifikante Unterschied ist, dass bei `getfa` und `putfa` die Indexwerte auf dem Stack liegen, und nicht als Immediate Werte an die Funktion übergeben werden. Technisch ist der Zugriff ansonsten äquivalent zu Records.

Verbundobjekttyp: Array (VM)

Um zur Laufzeit über alle Elemente eines Arrays zu iterieren benötigt man die Anzahl der im Verbundobjekt enthaltenen Elemente.

- In Ninja gibt es dafür die Anweisung: `sizeof(object)`
  - Wenn `object` ein **primitives Objekt** ist, wird `-1` zurückgegeben
  - Wenn `object` ein **Verbundobjekt** ist, wird die Anzahl der verwalteten Elemente (`ObjRef`) zurückgegeben

VM-Ebene Instruktion: `getsz ... object -> ... size` — Rückgabe soll wie angegeben erfolgen (`-1` bei primitiven Objekten, `object→size` bei Verbundobjekten).

Die `sizeof` -Anweisung in Ninja ist anders als in `C` . In Ninja erfolgt die Auswertung **dynamisch zur Laufzeit**, um z.B. die Größe eines Arrays zu bestimmten. In `C` erfolgt die Auswertung **statisch zur Kompilierzeit**, um Größen für Datentypen zu bestimmen.

Nil, Initialisierung, Laufzeittests

Jeder Referenz in Ninja ist zum Erzeugungszeitpunkt `nil` — Dies entspricht dem `NULL` -Zeiger in `C` oder `null` in `Java`. Dies signalisiert, dass ein Objekt nicht vorhanden ist!

Alle initialen Werte (lokale/globale Variablen, RVR, Komponenten vom BIP, Komponenten von Verbundobjekten) — *oder anders gesagt alle Rechenobjekte die mittels Referenz in den Heap dargestellt werden* — sind zu Beginn `nil`.

Dies erlaubt uns zur Laufzeit zu prüfen, ob auf ein gültiges Rechenobjekt zugegriffen wird oder nicht. In den meisten Fällen ist der Zugriff auf Objekte mit dem Wert `nil` ungültig → Man kann bzw. soll dann die Ausführung der VM abbrechen!

Nil, Initialisierung, Laufzeittests

Weiterhin soll ein Abbruch der VM-Ausführung erfolgen, wenn auf Felder in Arrays zugegriffen wird, die außerhalb der definierten Grenzen des Arrays liegen. D.h. wenn bei `(...)[index]` der `0 <= index < size` ist.

Generell kann diese Prüfung bei **Records** und **Arrays** verwendet werden. Wenn die Records allerdings vom Ninja-Compiler erzeugt wurden, sollten hier keine Fehlzugriffe auftreten!

Referenzvergleiche

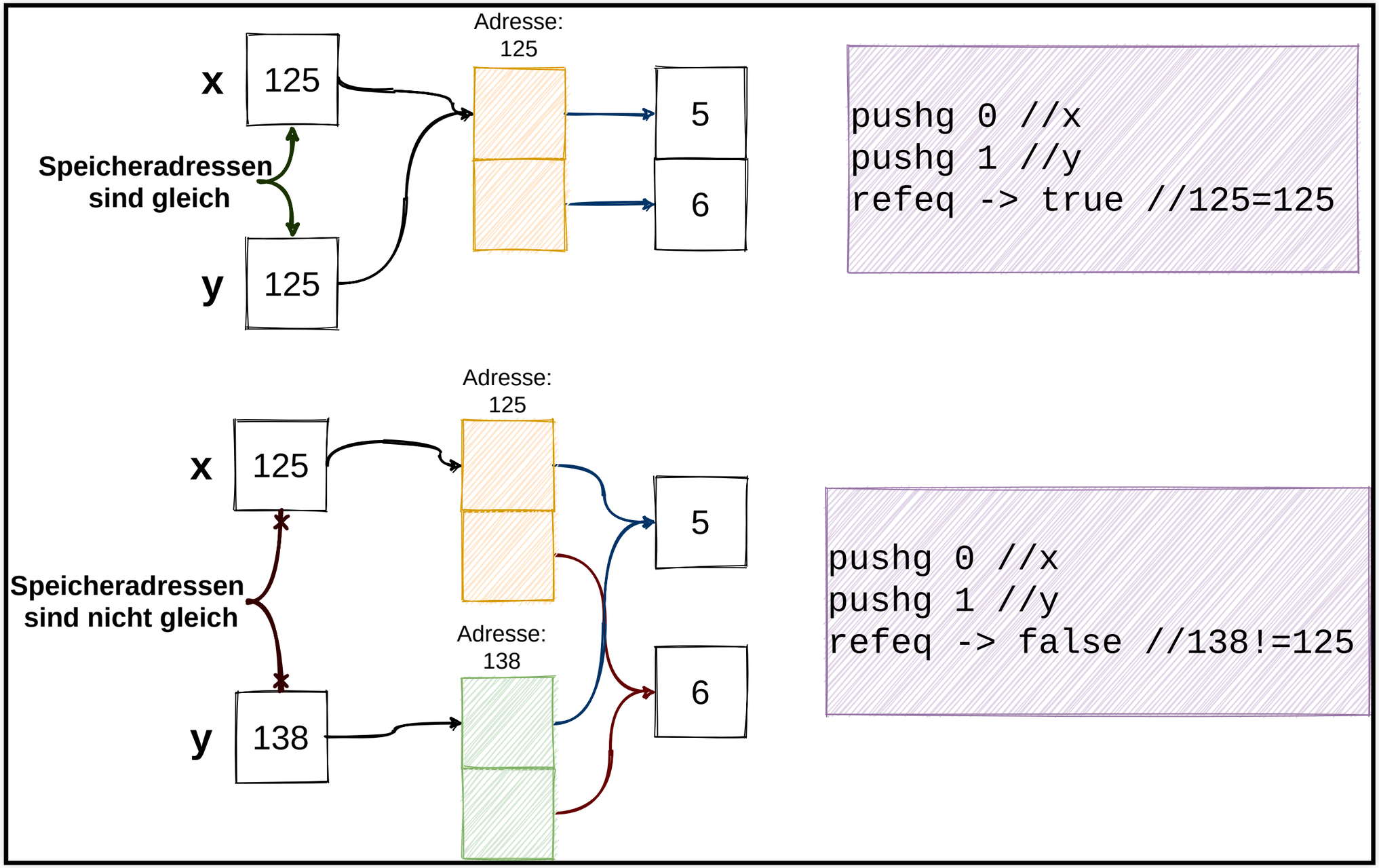
In Ninja kann ein Referenzvergleich durchgeführt werden (z.B. `x == nil`):

Hierzu werden verschiedene Instruktionen benötigt:

- `pushn ... -> ... nil` — legt ein `nil` auf den Stack
- `refeq ... x y -> ... b` — legt True (also `1`) auf den Stack, falls beide Objektreferenzen **gleich** sind (auf das gleiche Objekt zeigen!)
- `refne ... x y -> ... b` — legt True (also `1`) auf den Stack, falls beide Objektreferenzen **nicht gleich** sind (nicht auf das gleiche Objekt zeigen!)

Die VM verwendet zur Realisation der Vergleiche mit und auf `nil` den `NULL`-Pointer von `C`.

Referenzvergleiche



Abschluss

Hiermit können nun alle Aufgaben, inklusive der Aufgabe 7, implementiert werden.