KONZEPTE SYSTEMNAHER PROGRAMMIERUNG

Technische Hochschule Mittelhessen

Andre Rein

– Verbunde und Typdefinitionen –

REKAPITULATION UND MOTIVATION

Wir haben in Ninja bis jetzt verschiedene Rechenobjekte zur Verfügung: **Integer**, **Character** und **Boolean**, die jedoch alle auf **VM-Ebene** auf dem Stack als **ganzzahlige** Werte verwaltet werden.

Die Rechenobjekte werden implizit durch Ninja selbst unterschiedlich angelegt und behandelt:

- Einlesen Character → `rdchr`, Ausgabe Character `wrchr`
- Einlesen Integer → `rdint`, Ausgabe Integer `wrint`

Weiterhin liegen auf dem Stack nicht nur Rechenobjekte, sondern auch Verwaltungsdaten, wie z.B. **Rücksprungadresse**, **Framepointer**, etc.

Unsere Ninja-VM verwendet **intern** auf dem **Stack** bisher für alles den C-Datentyp `int`. Bisher war dies ausreichend.

REKAPITULATION UND MOTIVATION

- **Anforderung 1**: Zukünftig wollen wir **zusammengesetzte Objekte** unterstützen, z.B.:
  - *Ninja-Arrays*
  - *Ninja-Records*
- **Anforderung 2**: Zukünftig wollen wir die **Lebensdauer** unserer Rechenobjekte erhöhen, da der Zugriff auf diese Objekte oftmals über die **Ausführungsdauer** einer *Ninja-Prozedur* oder *Ninja-Funktion* hinausgeht.
  - Wir benötigen hierfür einen **Heap**-Speicher, auf dem Rechenobjekte beliebig lange (Ausführungszeit des Programms) gespeichert werden können

	*Globale Variablen* eigenen sich für die Erhöhung der Lebensdauer nur sehr eingeschränkt — hiermit kann keine sinnvolle dynamische Speicherverwaltung realisiert werden, die ggf. tausende oder hundert-tausende Rechenobjekte speichert und verwaltet.

REKAPITULATION UND MOTIVATION

Die Konsequenz aus den beiden Anforderungen: 1. **zusammengesetzte Objekte** und 2. **Lebensdauer dynamischer Rechenobjekte** ist, dass wir auf dem Stack nicht länger **ganzzahlige** Rechenobjekte und Verwaltungsdaten direkt verwenden können. Zukünftig werden wir auf dem Stack zwischen *Verwaltungsdaten* und *Rechenobjekten* unterscheiden müssen.

- Rechenobjekte werden zukünftig ausschließlich in einem Heap-Speicher gehalten — Auf dem Stack befinden sich nur noch **Referenzen** auf diese Objekte im Heap
- Verwaltungsdaten bleiben als ganzzahlige Werte auf dem Stack, müssen aber gesondert verwaltet und behandelt werden.

Für beides benötigen wir angepasste und eigene Datentypen.

TYPDEFINITIONEN

Frage: Wie kann man in `C` neue Datentypen definieren?

Genau wie man eine Variable definiert, jedoch mit dem vorangestellten Schlüsselwort `typedef`!

- `unsigned int U32;` definiert eine **Variable** mit dem Namen `U32` vom Typ `unsigned int`
- `typedef unsigned int U32;` definiert einen **Datentyp** mit dem Namen `U32` als Alias für den Typ `unsigned int`
  - Eine Variable `counter` vom Typ `U32` wird angelegt mit: `U32 counter;`
- `char * Messages[10];` definiert eine **Variable** mit Namen **Messages** (*ein Array mit 10 Zeigern auf char*)
- `typedef char * Messages[10];` definiert den **Typ** mit Namen **Messages** als Alias für ein Array mit 10 Zeigern auf char
  - Eine Variable `messages` vom Typ `Messages` wird angelegt mit: `Messages messages;`

TYPDEFINITIONEN: KONVENTIONEN

Typnamen beginnen *oftmals* mit einem **Großbuchstaben** oder *enden* mit einem `_t`.

- Beispiele:
  - Erster Buchstabe ist Großbuchstabe: `typedef unsigned int Size`
  - Angehängtes `_t` `typedef unsigned int size_t`

Somit lässt sich einfach und direkt erkennen, ob es sich bei einem *Namen* um eine **Variable** oder einen **Typ** handelt.

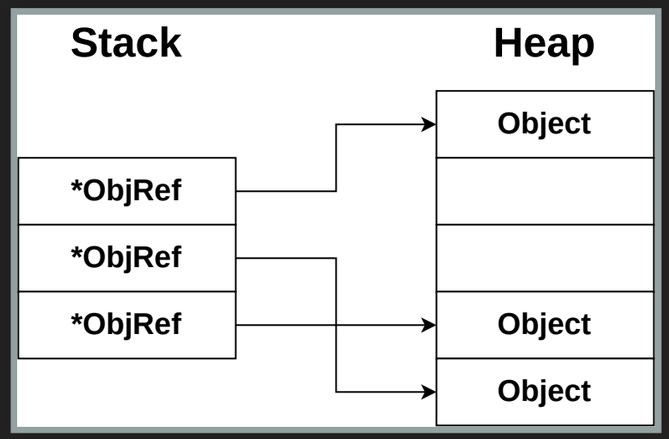
Den Typ `size_t` haben wir schon bei diversen Funktionen als Parameter und/oder Rückgabewert gesehen, bspw.(`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`). Es handelt sich hierbei um einen vordefinierten Typ, der per Definition einer vorzeichenlosen Ganzzahl (`unsigned int`) entspricht.

TYPDEFINITIONEN: OBJEKTE UND OBJEKTREFERENZEN

Wir sind nun in der Lage, neue Typen zu definieren. Unser Ziel ist es, Objekte im Heap und Referenzen auf diese Objekte auf dem Stack zu speichern.

Vorschlag:

```c
typedef int Object; /* Objekt */
typedef Object *ObjRef; /* Objektreferenz */
```



- **Vorteil:** Der eigentliche Typ von `Object` (d.h. `int`) kann später geändert werden, ohne dass sich Änderungen an der API, d.h. an unseren Funktionsprototypen, ergeben. — Die Implementierung muss immer noch angepasst werden, aber konsistente APIs für Funktionsaufrufe sind grundsätzlich sehr wünschenswert.

VERBUNDE (RECORDS):

Wir haben jedoch noch ein paar Probleme, die wir lösen müssen.

- **Problem 1**: Der Stack muss neben Rechenobjekten auch reine Zahlen aufnehmen (u.a. Framepointer **fp** und Returnadresse **ra**).
    - *Wie kann man in C den gleichen Speicher für Daten verschiedenen Typs (zu verschiedenen Zeiten nutzen)?*
- **Problem 2:** Die Rechenobjekte werden in Zukunft unterschiedlich (beliebig) **groß** sein (Anm: es wird nicht bei `int` bleiben) — Hierzu ist u.a. eine Größenangabe im Objekt notwendig.
    - *Wie macht man das in C?*

Beide Probleme können wir mit sog. *Verbunden* (engl. **Records**) lösen. Es gibt 2 Ausprägungen: *Fixe Verbunde* (sog. `struct`s) und *variable Verbunde* (sog. `union`s)

VERBUNDE (STRUCTS):

Ein Verbund (**Record**) ist eine Kollektion von Daten mit möglicherweise unterschiedlichen Datentypen. Die Auswahl einer sog. **Komponente** erfolgt durch den Namen.

- **Typ 1**: Fixe Verbunde (**structs**) haben Platz für alle der aufgezählten Komponenten

```
struct {
        char name[50];
        int tag;
        int monat;
        int jahr;
} person;
```

- Definiert eine Variable `person` mit 4 Komponenten (`name`, `tag`, `monat`, `jahr`)
- Die Auswahl der Komponente erfolgt über den Punkt-Operator `.`
  - Bspw. `person.jahr = 1979;`

STRUCTS:

Sehen Sie sich das eben angegebene Beispiel an. Wie viel Speicherplatz wird eine Variable `person` hier wohl benötigen?

```
struct {
    char name[50];
    int tag;
    int monat;
    int jahr;
} person;
```

- **Antwort 1:** `50` Byte (`char name[50]`) + `12` Byte (3 * 4 `int`) = `62` Byte.
- **Antwort 2:** Einigen wir uns darauf, dass in dem o.a. Fall **mindestens** `62` Byte reserviert werden. Wie viel Byte der Compiler tatsächlich anlegt, liegt u.a. am Compiler selbst und der Rechnerarchitektur. (Mit `gcc` auf X86_64 wurden `64` Byte verwendet!)
  - *Prüfung auf Speichergröße zur Laufzeit möglich mit:* `sizeof(person);`

STRUCTS - SPEICHERGRÖSSE:

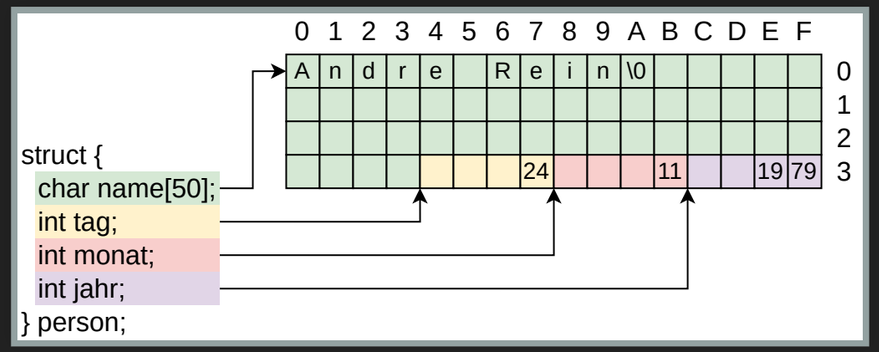
Frage: *Warum werden auf der X86_64 Architektur 64 Byte (und nicht 62 Byte) für das vorherige Beispiel reserviert?*

- Ein Datenwort entspricht auf der X86_64 Bit Architektur **4 Byte**. Dies ist die kleinste Einheit, die der Prozessor einlesen kann. *Selbst wenn nur ein Byte gelesen werden soll, liest der Prozessor zuerst 4 Byte aus und verwirft ggf. die nicht benötigten Daten.* Der Compiler ordnet nun die Datenstrukturen im Speicher so an, dass möglichst effizient darauf zugegriffen werden kann!
- Diese Optimierung kann man bei Bedarf mit den Schlüsselworten `__attribute__((__packed__))` abschalten, der Zugriff wird dann aber **langsamer**!

STRUCT-BEISPIEL:

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

int main(int argc, char *argv[]) {
    struct {
        char name[50];
        int tag;
        int monat;
        int jahr;
    } person;

    printf("struct person -- size [%lu] Bytes \n",
            sizeof(person));
    strcpy(person.name, "Andre Rein");
    person.tag   = 24;
    person.monat = 11;
    person.jahr  = 1979;

    printf("Person -- Name (\"%s\")\n", person.name);
    printf("        - Geburtsdatum (%2d.%2d.%4d)\n",
            person.tag,
            person.monat,
            person.jahr);

    return 0;
}
```

```
$ gcc -g -Wall -pedantic person.c -o person
$ ./person
struct person -- size [64] Bytes
Person -- Name ("Andre Rein")
        - Geburtsdatum (24.11.1979)
```



VARIANTE VERBUNDE (UNIONS)

- **Typ 2**: Variante Verbunde (**unions**) haben Platz für eine der aufgezählten Komponenten. Und zwar für die, die den meisten Speicherplatz benötigt
  - Daraus folgt, dass im Speicherplatz der Komponente auch Daten gespeichert werden können, die weniger Speicherplatz beanspruchen!

```c
union {
    double d;
    unsigned char b[sizeof (double)];
} inspect;
```

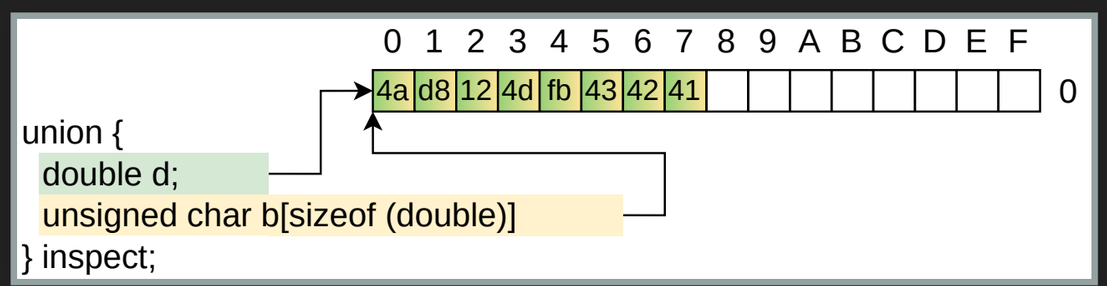
Definiert eine Verbund-Variable `inspect`, die Platz für *entweder* eine double-Größe *oder* ein entsprechend großes Byte-Array hat.

	*Im o.a. Fall sind beide Komponenten gleich groß, das muss aber nicht sein!*

UNION-BEISPIEL:

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

int main(int argc, char *argv[]) {
    union {
        double d;
        char b[sizeof(double)];
    } inspect;

    printf("union inspect -- size [%lu] Bytes \n",
           sizeof(inspect));
    inspect.d   = 3.1415926;
    printf("inspect -- Value (%f)\n", inspect.d);
    for (int i=0; i < sizeof(double); i++){
        printf("[%d] = 0x%02hhx\n", i, inspect.b[i]);
    }
    puts("");
    inspect.b[7]   = 'A';
    inspect.b[6]   = 'B';
    inspect.b[5]   = 'C';
    printf("inspect -- Value (%f)\n", inspect.d);
    for (int i=0; i < sizeof(double); i++){
        printf("[%d] = 0x%02hhx\n", i, inspect.b[i]);
    }
    puts("");
    return 0;
}
```

```
$ gcc -g -Wall -pedantic inspect.c -o inspect
$ ./inspect
union inspect -- size [8] Bytes
inspect -- Value (3.141593)
[0] = 0x4a
[1] = 0xd8
[2] = 0x12
[3] = 0x4d
[4] = 0xfb
[5] = 0x21
[6] = 0x09
[7] = 0x40

inspect -- Value (2394102.602138)
[0] = 0x4a
[1] = 0xd8
[2] = 0x12
[3] = 0x4d
[4] = 0xfb
[5] = 0x43
[6] = 0x42
[7] = 0x41
```



VARIANTE VERBUNDE (UNIONS)

Aus der Perspektive des Speichers wird für eine `union` einfach Speicherplatz reserviert. Die Größe entspricht hierbei der größten Komponente der Union. Der Compiler weiß (anhand des Quellcodes) auf welche Komponente und damit auf welchen Datentyp zugegriffen wird. Das bedeutet, der Wert wird dem Zugriff entsprechend **interpretiert**.

VARIANTE VERBUNDE (UNIONS)

Beispiel mit unterschiedlich großen Datentypen

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


int main(int argc, char *argv[]) {
    union {
        int i;    // 4 Byte
        char c;   // 1 Byte
    } test;

    printf("union test - size [%lu] Bytes \n", sizeof(test));
    test.i = 0x41424344;
    printf("test.i [int]  -> (%d) [0x%0x]\n", test.i, test.i);
    printf("test.c [char] -> (%c) [0x%0x]\n", test.c, test.c);

    return 0;
}
```

```
$ gcc -g -Wall -pedantic number2.c -o number2
$ ./number2
union test - size [4] Bytes
test.i [int]  -> (1094861636) [0x41424344]
test.c [char] -> (D) [0x44]
```

TYPDEFINITION UND VERBUNDE

Oft wollen wir nicht eine einzige Variable eines Verbund-Typs erstellen, sondern mehrere.

Beispielsweise wollen wir gerne einen Typ `Person` definieren, mit dessen Hilfe wir nun mehrere Variablen anlegen können, die unterschiedliche Personen beschreiben.

```
typedef struct {        1
    char name[50];
    int tag;
    int monat;
    int jahr;
} Person;               1

Person person_1, person_2; 2
```

1. Typdefinition
2. Anlegen von 2 Variablen vom Typ Person

TYPDEFINITION UND VERBUNDE

Rekursive Strukturen, also Strukturen die sich selbst enthalten, lassen sich so jedoch nicht abbilden. Als klassisches Beispiel könnte hier eine einfach verkettete Liste dienen:

```
typedef struct {
    int number;
    Liste *next; (1)
} Liste;

Liste myliste;
```

(1) Der Typ Liste wird verwendet, bevor er definiert ist.

Dafür gibt es sog. **Tagnames**, die Teil der Typdefinition werden.

TYPDEFINITION UND VERBUNDE

Um rekursive Strukturen wie verkettete Listen oder Bäume zu ermöglichen, wird bei Verbunden ein sog. **Tagname** als Teil der Definition eingeführt.

```
typedef struct liste { (1)
    int number;
    struct liste* next; (2)
} Liste;

Liste mylist;
```

(1) Einführen des Tagnames `liste`

(2) Der Typ `struct liste` wird verwendet für die rekursive Struktur

	Rekursive Strukturen können nur mittels Zeigern abgebildet werden. D.h., dass bei der o.a. Verwendung der Typ `struct liste` nicht direkt verwendet werden kann. Es muss ein Zeiger auf diesen Typ (`struct liste*`) verwendet werden!

TYPDEFINITION UND VERBUNDE: ANMERKUNG

Die Typdefintion mittels `typedef` spart in folgenden Verwendungsschritten, z.B. beim Anlegen der Variablen, das keyword `struct`.

Man kann jedoch auch gänzlich ohne `typedef` arbeiten, muss dann allerdings immer das Schlüsselwort `struct`, wie bei der rekursiven Definition (`struct liste* next;`) verwenden.

```c
struct list { 1
    int number;
    struct list* next; 2
};

struct list mylist;
```

Welche Variante sie hier verwenden, ist Ihre Entscheidung.

TYPDEFINITION UND VERBUNDE: ANMERKUNG

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    typedef struct liste {
        int number;
        struct liste *next;
    } Liste;

    Liste mylist;

    printf("mylist  -- size [%d] Bytes\n", sizeof(Liste));

    struct list {
        int number;
        struct list *next;
    };

    struct list mylist2;
    printf("mylist2 -- size [%d] Bytes\n", sizeof(struct list));

    return 0;
}
```

```
$ gcc -g -Wall -pedantic list.c -o list
$ ./list
mylist  -- size [16] Bytes
mylist2 -- size [16] Bytes
```

ZUGRIFFE AUF VERBUNDE (INSB. STRUCTS)

Der Zugriff auf Komponenten in structs erfolgt mit dem Punkt-Operator `.`. Z.B. `person.monat`.

Oft ist es aber so, das Verbunde dynamisch angelegt werden, d.h. das die Speicherplatzreservierung der angelegten Variablen zur Laufzeit mit `malloc` erfolgt.

In diesem Fall kann man auf die Komponenten nicht mehr einfach mit dem `.`-Operator zugreifen. Hierzu müsste die angelegte Variable erst dereferenziert werden.

Da diese Art von Zugriffen sehr gebräuchlich ist, gibt es den Operator `->`, der zuerst eine Dereferenzierung und anschließend einen Zugriff auf die Komponente durchführt.

ZUGRIFFE AUF VERBUNDE BEISPIEL

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

int main(int argc, char *argv[]) {
    typedef struct person {
        char name[50];
        int tag;
        int monat;
        int jahr;
    } Person;

    Person *p1;
    p1=malloc(sizeof(Person));
    //p1.monat=10;                          1
    strcpy((*p1).name, "Andre Rein");       2
    (*p1).tag=24;                           2
    p1->monat=11;                           3
    p1->jahr=1979;                          3

    printf("Person -- Name (\"%s\")\n", p1->name);  3
    printf("        - Geburtsdatum (%2d.%2d.%4d)\n",
           p1->tag,                         3
           (*p1).monat,                     2
           (*p1).jahr);                     2
    return 0;
}
```

1. Ungültiger Zugriff mittels `.`
2. **Gültiger** Zugriff nach **expliziter** Dereferenzierung mittels `.`

   (Achtung: Wegen Operator-Prioritäten muss geklammert werden!)
3. **Gültiger** Zugriff nach **impliziter** Dereferenzierung mittels `->`

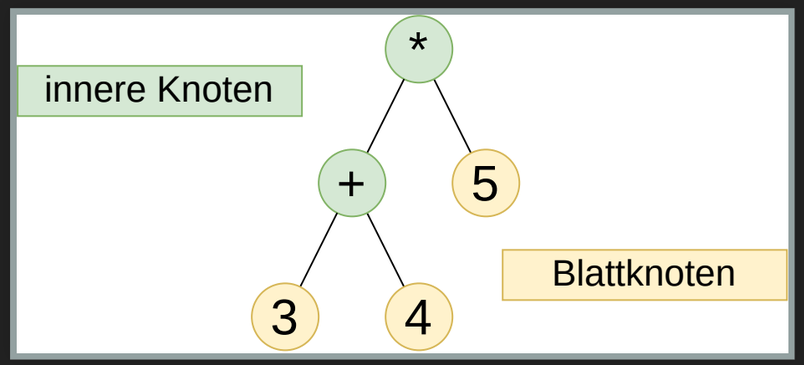
```
$ gcc -g -Wall -pedantic person_pointer.c -o person_p
$ ./person_p
Person -- Name ("Andre Rein")
        - Geburtsdatum (24.11.1979)
```

ℹ	Die beiden Ausdrücke `(*p1).xxx` und `p1->xxx` sind äquivalent!

ANWENDUNGSBEISPIEL

Baumstruktur für Arithmetische Ausdrücke

Beispiel: Darstellung und Auswertung des Arithmetischen Ausdrucks `(3+4)*5`



- **Innerer Knoten**:
  - Arithmetische Operationen
  - Linker Teilbaum
  - Rechter Teilbaum
- **Blattknoten**:
  - Zahl

	Ein Knoten ist **entweder** ein *innerer Knoten* **oder** ein *Blattknoten*, d.h. wir verwenden hierzu eine `union`

ANWENDUNGSBEISPIEL

tree.h

```h
#ifndef TREE_H
#define TREE_H

typedef struct node {
    char isLeaf;
    union {
        struct {
            char operation;
            struct node *left;
            struct node *right;
        } innerNode;
        int value;
    } u;
} Node;

Node *newLeafNode(int value);
Node *newInnerNode(char operation, Node* left, Node* right);

#endif  /*TREE_H*/
```

tree.c

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

Node *newLeafNode(int value){
    Node *n;
    if ((n = malloc(sizeof(Node))) == NULL) {
        perror("malloc");
    }
    n->isLeaf=1;
    n->u.value=value;
    return n;
}

Node *newInnerNode(char operation, Node* left, Node* right){
    Node *n;
    if ((n = malloc(sizeof(Node))) == NULL) {
        perror("malloc");
    }
    n->isLeaf=0;
    n->u.innerNode.operation=operation;
    n->u.innerNode.left=left;
    n->u.innerNode.right=right;
    return n;
}
```

ANWENDUNGSBEISPIEL

main.c

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


int eval(Node *);
void freeTree(Node *);

int main(int argc, char *argv[]) {
    Node *expression;
    int result;

    expression = newInnerNode(
            '*',
            newInnerNode('+',
                newLeafNode(3),
                newLeafNode(4)),
            newLeafNode(5)
            );

    result = eval(expression);
    printf("(3+4)*5 = %d\n", result);
    freeTree(expression);
    return 0;
}

int eval(Node *tree) {
    int op1, op2, result;

    if (tree->isLeaf) {
        result = tree->u.value;
    } else {
        op1 = eval(tree->u.innerNode.left);
        op2 = eval(tree->u.innerNode.right);
        switch (tree->u.innerNode.operation){
```

main.c

```c
            case '+':
                result = op1 + op2;
                break;
            case '-':
                result = op1 - op2;
                break;
            case '*':
                result = op1 * op2;
                break;
            case '/':
                result = op1 / op2;
                break;
            default:
                printf("Error - operation not implemented\n");
                exit(1);
        }
    }
    return result;
}

void freeTree(Node *tree) {
    if (tree->isLeaf) {
        free(tree);
    } else {
        freeTree(tree->u.innerNode.left);
        freeTree(tree->u.innerNode.right);
        free(tree);
    }
}
```

```
$ gcc -g -Wall -pedantic -std=c99 main.c tree.c -o express
$ ./express
(3+4)*5 = 35
```

RELEVANZ FÜR NINJAVM

Unser Ziel ist es, die ehemaligen Rechenobjekte auf dem Stack durch Rechenobjekte auf dem Heap zu ersetzen. Hierzu verwalten wir auf unserem Stack nun Objektreferenzen (Zeiger auf die Rechenobjekte).

```
typedef int Object; /* Rechenobjekt */
typedef Object *ObjRef; /* Referenz auf ein Rechenobjekt*/
```

Jedoch gibt es auf dem Stack auch Objekte, die keine Rechenobjekte sind. Z.B. die Returnadresse (`ra`) und die Position des alten Framepointers (`old fp`) bei Funktionsaufrufen. Diese Objekte sollen weiterhin direkt auf dem Stack verwaltet werden!

Hierzu definieren wir uns nun einen neuen Typ `StackSlot`, der mit Hilfe von `struct` und `union` eine Unterscheidung zwischen Rechenobjekten (im Heap) und anderen Objekten (auf unserem Stack) erlaubt.

STACKSLOTS

```c
#include <stdbool.h>

typedef int Object;
typedef Object *ObjRef;

typedef struct {
        bool isObjRef;
        union {
                ObjRef objRef; // isObjRef = TRUE
                int number;    // isObjRef = FALSE
        } u;
} StackSlot;

StackSlot stack[STACK_SIZE]; // ehemals int stack[STACK_SIZE]
```

Der Zugriff erfolgt nun je nachdem wie `isObjRef` ausgewertet wird:

- `isObjRef == true`: `stack[sp].u.ObjRef`
- `isObjRef == false`: `stack[sp].u.number`

STATIC DATA AREA UND RETURN REGISTER

Das Register für Rückgabewerte (**RVR**) und die Static Data Area (**SDA**) müssen auch angepasst werden und sollen zukünftig nur noch Objektreferenzen verwalten. Da im RVR und der SDA ausschließlich Rechenobjekte enthalten sind, muss hier keine Unterscheidung wie auf dem Stack getroffen werden.

```c
#include <stdbool.h>

typedef int Object;
typedef Object *ObjRef;

// SDA
size_t sda_size = number_global_vars * sizeof(ObjectRef); // 8 Byte
ObjRef *sda = malloc(sda_size); // array - sda_size is known!

// RVR

ObjRef rv; // single object
```

SPEICHERN VON OBJEKTEN

Bisher verwenden wir zur Repräsentation von Objekten noch ordinäre Ganzzahlen (`int`). Wir werden allerdings zukünftig verschiedene Objekttypen verwalten müssen, die u.U. beliebig groß werden können.

Wir benötigen also eine eigene Datenstruktur, die wir flexibel an unsere jeweiligen Bedürfnisse anpassen können.

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

typedef Object *ObjRef;
```

Zukünftig werden wir nur den Typ `ObjRef` verwenden. Der Typ `Object` wird nicht mehr benötigt. Alle zukünftigen Zugriffe erfolgen ausschließlich über Referenzen und wir können beliebige Objekte im Typ `ObjRef` abbilden.

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

INTEGER ALS OBJREF

Bisher sind jedoch weiterhin alle unsere Rechenobjekte Integerwerte. Das heißt, wir müssen nun einen Weg finden, wie wir unsere Integerwerte in einem Objekt vom Typ `ObjRef` anlegen und speichern können.

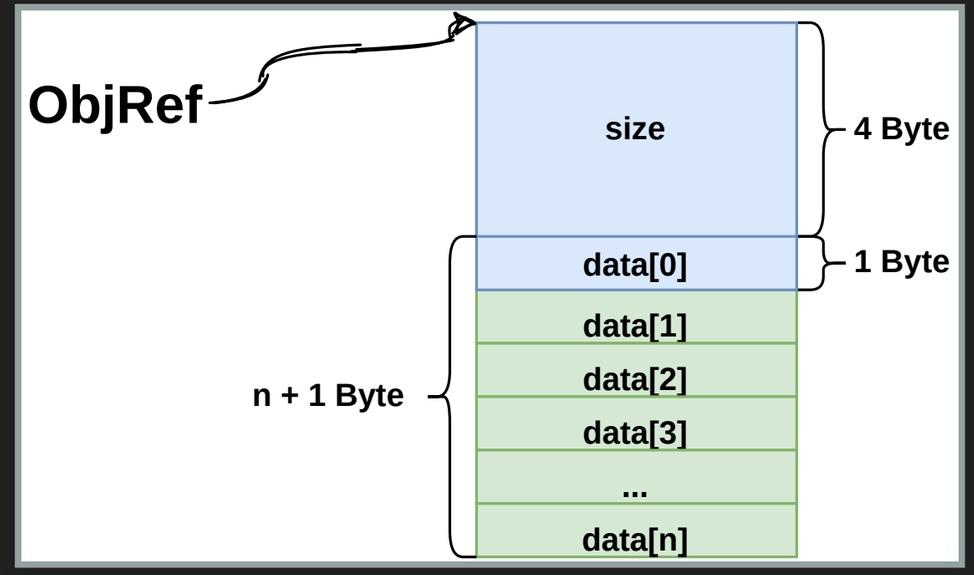
Umsetzen des Konzeptes `size as needed` mit `malloc()`

```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 intObject;
    unsigned int objSize = sizeof(unsigned int) + sizeof(int);
    if ((intObject = malloc(objSize)) == NULL) {
        perror("malloc");
    }

    intObject->size=sizeof(int);
    *(int *)intObject->data = 5;
    printf("intObject->data = %d\n", *(int *)intObject->data);
    free(intObject);
    return 0;
}

// Ausgabe: intObject->data = 5
```

INTEGER ALS OBJREF: SIZE AS NEEDED

Unsere Definition von `ObjRef`, legt eine Struktur mit der (minimal-) Größe `5` ** an. Da in C keine Bereichsprüfung durchgeführt wird, kann beliebig auf reservierten Speicher zugegriffen werden. `malloc` reserviert für uns zusammenhängenden Speicher; deshalb können wir beliebig viel Speicher anfordern und uns so beliebig große **konkrete** Datentypen anlegen!

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

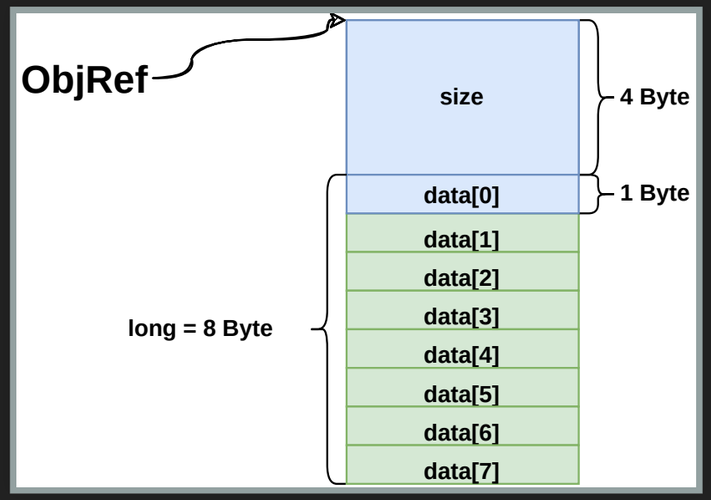


⚠	** Die Größe 5 bedeutet hier, dass mindestens `5` Byte von `ObjRef` belegt werden. Der Compiler wird hier jedoch i.d.R. immer `8` Byte auf X86_64 anlegen.

BEISPIEL - LONG ALS OBJREF: SIZE AS NEEDED

```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 longObject;
    unsigned int objSize = sizeof(*longObject) + sizeof(long);
    if ((longObject = malloc(objSize)) == NULL) {
        perror("malloc");
    }

    longObject->size=sizeof(long);
    *(long *)longObject->data = 5;
    printf("longObject->size = %d\n", longObject->size);
    printf("longObject->data = %ld\n", *(long *)longObject->data);
    printf("*longObject size = %lu\n", sizeof(*longObject));
    free(longObject);
    return 0;
}
```

```
$ gcc -g -Wall -pedantic -std=c99 objref2.c -o objref2
$ ./objref2
longObject->size = 8
longObject->data = 5
*longObject size = 8 // FALSCH! Es sind 12 Byte!
```



Die Objektgröße wird mit `sizeof()` als 8 angegeben. Das ist aber Falsch. Das Objekt ist 12 Byte (4 int (`size`) + 8 long (`data`)).

BEISPIEL - LONG ALS OBJREF: SIZE AS NEEDED

Wie wir gesehen haben, können wir uns nicht mehr auf die Funktion `sizeof()` verlassen, um die Größe unserer Objekte zu bestimmten. Die Größe unserer Objekte ist dem Compiler **unbekannt** und muss von uns vollständig **selbst verwaltet** werden!

Jedoch funktioniert eine explizite Prüfung nur dann, wenn wir selbst die Größe kennen bzw. mit Datentypen arbeiten, die eine bekannte und feste Größe haben. Ansonsten müssen wir uns darauf verlassen, was in `objRef->size` angegeben ist!

Wenn bekannte primitive Datentypen, wie `int` oder `long`, verwendet werden, kann bei Zuweisungen geprüft werden, ob die Größe des Zielspeichers (`data`) der Größe entspricht, die wir erwarten.

```
int x;
if (objRef->size == sizeof(int)) {
        x = *(int *) objRef->data;
} else {
        printf("Error, not enough space in object!");
        exit(1);
}
```

UMSTELLUNG AUF OBJEKTREFERENZEN

Wir können nun die Umstellung auf Objektreferenzen, wie in Aufgabe 5 gefordert, vollständig umsetzen. Hierzu müssen alle Funktionen (Prototypen und Implementierung) angepasst werden, damit der Stack, das Rückgaberegister (RVR) und die Static Data Area (SDA) Objektreferenzen verwenden können.

Im Zuge dessen muss es weiterhin möglich sein, Verwaltungsdaten, wie z.B. Framepointer und Returnadressen, direkt auf dem Stack zu speichern und zu verarbeiten. Dies sind keine Rechenobjekte!