KONZEPTE SYSTEMNAHER PROGRAMMIERUNG

Technische Hochschule Mittelhessen

Andre Rein

– Laden von Ninja Code und Speicherverwaltung mit Zeigern –

NINJA-CODE AUS DATEIEN LADEN

- Bis jetzt haben wir die Instruktionen unserer Programme direkt in unserer VM hinterlegt
    - Hierzu haben wir **Bytecode** direkt in internen Datenstrukturen (*Arrays*) abgelegt und ausgeführt

Diese Vorgehensweise ist natürlich ineffektiv, da wir jede Anweisung (z.B. ein Berechnung, Eingabe, Ausgabe) bis jetzt so zu sagen "per Hand" in Ninja Bytecode umwandeln mussten. Aus Programmiersicht verlassen wir nun die Ebene der direkten Bytecode-Erzeugung und wenden uns der Ebene der Ninja Assembler-Programmierung zu. Die Übersetzung zwischen Ninja Assemblercode und Ninja Bytecode übernimmt ab sofort der **Ninja Assembler** `nja`.

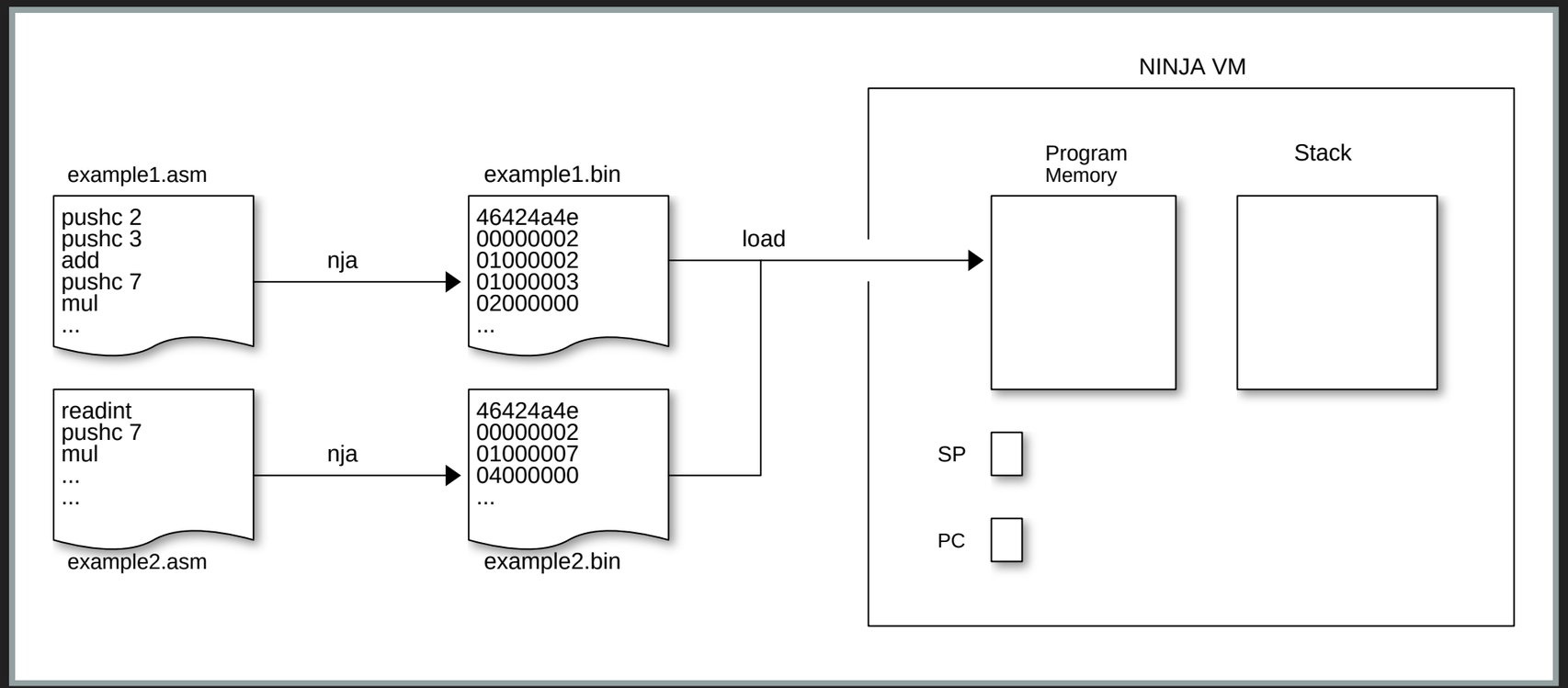
NINJA-ASSEMBLER

- Die Programmierung erfolgt nun also auf der Ebene von Ninja Assemblercode
  - Der Ninja Assemblercode wird hierbei typischerweise in einer Datei mit der Endung `.asm` abgespeichert
- Der Ninja Assembler (`nja`) erhält als Eingabe eine Datei mit Ninja Assemblercode (z.B. `example.asm`) und
- erzeugt als Ausgabe eine Datei in Ninja Bytecode mit der Endung `bin` (z.B. `example.bin`)

```
//
// example.asm
//
// (2 + 3) *7;
// writeInteger();
// writeCharacter('\n');

pushc   2
pushc   3
add
pushc   7
mul
wrint
pushc   '\n'
wrchr
halt
```

```
$ ./nja example.asm example.bin
$ xxd -e example.bin
00000000: 46424a4e 00000002 00000009 00000000  NJBF............
00000010: 01000002 01000003 02000000 01000007  ................
00000020: 04000000 08000000 0100000a 0a000000  ................
00000030: 00000000                             ....
$ ./njvm example.bin
Ninja Virtual Machine started
000:    pushc   2
001:    pushc   3
002:    add
003:    pushc   7
004:    mul
005:    wrint
006:    pushc   10
007:    wrchr
008:    halt
35
Ninja Virtual Machine stopped
```

NINJA VM ÜBERSICHT



Die vom Ninja Assembler `nja` erzeugten Dateien, die den Ninja Bytecode enthalten, können nun in den **Programmspeicher** der Ninja VM geladen und anschließend ausgeführt werden!

LADEN VON DATEIEN

Um eine Datei zu **öffnen** und in den Programmspeicher zu **laden**, benötigen wir bestimmte `C`-Funktionen.

- Öffnen von Dateien `fopen`
- Lesen von Inhalten einer Datei `fread`
- Positionierung zum Lesen und Schreiben `fseek`
- Schließen einer geöffneten Datei `fclose`

Folgende Beispiele werden exemplarisch an der Datei `test.bin` ausgeführt:

Dateiinhalt: `ABCDEFGHIJKLMNOPQRTSUVWXYZ0123456789`

Erzeugung der Datei:

```
$ echo -n "ABCDEFGHIJKLMNOPQRTSUVWXYZ0123456789" > test.bin
```

Anmerkung: *Bei diesem Dateiinhalt handelt es sich natürlich nicht um validen Ninja Bytecode!*

ÖFFNEN VON DATEIEN

```
FILE *fopen(const char *pathname, const char *mode);
```

- Pfad (`path`):
  - Absoluter oder relativer Pfad zu einer Datei
- Modi (`mode`):
  - `r` lesen, `r+` lesen und schreiben,
  - `w` / `w+` lesen/schreiben + Erzeugen (überschreiben),
  - `a` / `a+` lesen/schreiben+Erzeugen (anhängen)
- Rückgabe: Zeiger auf eine Verwaltungsstruktur einer Datei (`FILE *`) oder `NULL` im Fehlerfall

```c
#include <stdio.h>


int main(int argc, char *argv[]){
    FILE * fp=NULL;
    if ((fp = fopen("./test.bin", "r"))==NULL) { 1
        perror("ERROR - fopen"); 2
        exit(1); 2
    }
}
```

1. Öffnen der Datei `test.bin` im aktuellen Arbeitsverzeichnis
2. Programmende im Fehlerfall

LESEN VON INHALTEN

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Rückgabe: Anzahl der gelesenen Datenelementen (`nmemb`)
  - entspricht nur dann Bytes wenn `size` = `1`
  - im Fehlerfall **und** bei Dateiende wird `0` zurückgegeben!
    - Unterscheidung nur über Umweg möglich! `man 3 fread`
- Anmerkung Parameter `void *ptr`:
  - **Zeiger** auf beliebigen Speicherplatz → kann also auf beliebigen Datentyp zeigen!
    weitere Beispiele hierzu folgen später

```c
#include <stdio.h>

int main(int argc, char *argv[]){
    FILE * fp=NULL;
    if ((fp = fopen("./test.bin", "r"))==NULL) {
        perror("ERROR - fopen");
        exit(1);
    }
    char c[4]; 1
    read_len = fread(c, 1, 4, fp); 2
    printf("r %d bytes: c = [%c, %c, %c, %c]",
        read_len, c[0], c[1], c[2], c[3]); 3

    // AUSGABE: "r 4 bytes: c = [A, B, C, D]"
}
```

1. Array `c` mit Platz für 4 Bytes
2. Einlesen von `4x1` Byte in Array `c`
3. Ausgabe: c = [A, B, C, D]

LESEN VON DATEIINHALTEN:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Überlegen Sie kurz: Der Aufruf einer einzelnen Ninja-Instruktion (Bytecode) kann mit `fread` auf 2 unterschiedliche Arten erfolgen.
  - Welcher Rückgabewert wird hierbei `read_len` zugewiesen und welchen Aufruf halten Sie für sinnvoller?

- Gegeben sei: `unsigned int inst, read_len;` und ein valides `fp`.
  - `read_len = fread(&inst, 1, sizeof(unsigned int), fp);`
  - `read_len = fread(&inst, sizeof(unsigned int), 1, fp);`

POSITIONIERUNG ZUM LESEN UND SCHREIBEN

```
int fseek(FILE *stream, long offset, int whence);
```

- Modi (`whence`):
  - `SEEK_SET` Dateianfang,
  - `SEEK_CUR` aktuelle Position,
  - `SEEK_END` Dateiende
- Offset: Position in Bytes relativ zu `whence` (positiv oder negativ)
- Rückgabe: `0` wenn das Setzen funktioniert hat, im Fehlerfall `-1`

```c
#include <stdio.h>


int main(int argc, char *argv[]){
    FILE * fp=NULL;
    if ((fp = fopen("./test.bin", "r"))==NULL) {
        perror("ERROR - fopen");
        exit(1);
    }
    read_len = fread(c, 1, 4, fp); (1)
    fseek(fp, 2, SEEK_CUR) (2)
    fseek(fp, 0, SEEK_SET) (3)
    fseek(fp, 0, SEEK_END) (4)
}
```

1. Position → `4`
2. Position → `6`
3. Position → `0` (Dateianfang)
4. Position → ` ` (Dateiende)

SCHLIESSEN EINER GEÖFFNETEN DATEI

`int fclose(FILE *stream);`

- Rückgabe: `0`, falls das Schließen funktioniert hat
- im Fehlerfall Wert `!=0`

Nach dem Aufruf von `fclose(fp)` darf `fp` nicht mehr verwenden, da `fp` nun auf einen (*semantisch*) ungültigen Speicherbereich zeigt. Eine Verwendung von `fp` nach `fclose(fp)` bedeutet undefiniertes Verhalten, da der Speicher möglicherweise bereits anderweitig wiederverwendet wurde!

```c
#include <stdio.h>

int main(int argc, char *argv[]){
    FILE * fp=NULL;
    if ((fp = fopen("./test.bin", "r"))==NULL) {
        perror("ERROR - fopen");
        exit(1);
    }
    read_len = fread(c, 1, 4, fp);
    if (fclose(fp) !=0) { 1
        perror("ERROR - fclose"); 2
        exit(1); 2
    }
    3
}
```

1. Aktuell geöffnete Datei (referenziert durch `fp`) schließen
2. Programmende im Fehlerfall
3. `fp` nicht mehr verwenden!

VOLLSTÄNDIGES BEISPIEL

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

int main(int argc, char *argv[]){
    FILE * fp=NULL;
    char filename[128];
    char error_msg[256];
    int read_len=0;
    sprintf(filename, "./test.bin");
    if ((fp = fopen(filename, "r"))==NULL) {
        snprintf(error_msg, 256, "ERROR (fdopen) -> File (%s)",
                filename);
        perror(error_msg);
        exit(1);
    }
    char c[4];
    unsigned int x;
    printf("idx   HEX  CHAR |idx   HEX  CHAR |");
    printf("idx   HEX  CHAR |idx   HEX  CHAR\n");
    while ((read_len = fread(c, 1, 4, fp)) != 0) {
        printf("c[0]=[0x%1$x] [%1$c] |", c[0]);
        printf("c[1]=[0x%1$x] [%1$c] |", c[1]);
        printf("c[2]=[0x%1$x] [%1$c] |", c[2]);
        printf("c[3]=[0x%1$x] [%1$c]\n", c[3]);
    }
    printf("\nfile position = [%lu] after 1. loop\n", ftell(fp));
    fseek(fp, 0, SEEK_SET); // seek back to beginning of file
    while ((read_len=fread(&x,sizeof(unsigned int),1,fp)) != 0) {
        printf("read %d object [%ld bytes]: x = [0x%08x]\n",
                read_len, read_len*sizeof(unsigned int), x);
    }
    if (fclose(fp) != 0){
        perror("ERROR (fclose)");
    }
    return 0;
}
```

```
$ gcc file_handling.c -g -o file_handling && ./file_handling
idx   HEX  CHAR |idx   HEX  CHAR |idx   HEX  CHAR |idx   HEX  CHAR
c[0]=[0x41] [A] |c[1]=[0x42] [B] |c[2]=[0x43] [C] |c[3]=[0x44] [D]
c[0]=[0x45] [E] |c[1]=[0x46] [F] |c[2]=[0x47] [G] |c[3]=[0x48] [H]
c[0]=[0x49] [I] |c[1]=[0x4a] [J] |c[2]=[0x4b] [K] |c[3]=[0x4c] [L]
c[0]=[0x4d] [M] |c[1]=[0x4e] [N] |c[2]=[0x4f] [O] |c[3]=[0x50] [P]
c[0]=[0x51] [Q] |c[1]=[0x52] [R] |c[2]=[0x53] [S] |c[3]=[0x54] [T]
c[0]=[0x55] [U] |c[1]=[0x56] [V] |c[2]=[0x57] [W] |c[3]=[0x58] [X]
c[0]=[0x59] [Y] |c[1]=[0x5a] [Z] |c[2]=[0x30] [0] |c[3]=[0x31] [1]
c[0]=[0x32] [2] |c[1]=[0x33] [3] |c[2]=[0x34] [4] |c[3]=[0x35] [5]
c[0]=[0x36] [6] |c[1]=[0x37] [7] |c[2]=[0x38] [8] |c[3]=[0x39] [9]

file position = [36] after 1. loop
read 1 object [4 bytes]: x = [0x44434241]
read 1 object [4 bytes]: x = [0x48474645]
read 1 object [4 bytes]: x = [0x4c4b4a49]
read 1 object [4 bytes]: x = [0x504f4e4d]
read 1 object [4 bytes]: x = [0x54535251]
read 1 object [4 bytes]: x = [0x58575655]
read 1 object [4 bytes]: x = [0x31305a59]
read 1 object [4 bytes]: x = [0x35343332]
read 1 object [4 bytes]: x = [0x39383736]
$
```

Beachten Sie bei der zweiten Ausgabe, dass die Werte genau so ausgegeben werden, wie sie im Speicher vorhanden sind!

VOLLSTÄNDIGES BEISPIEL: ANMERKUNG

Die `while`-Schleife um `fread` wurde verwendet um beliebig viele einzelne Datenelemente einer Datei auszulesen (*die Anzahl an Elementen war hierbei unbekannt!*). Wenn bekannt ist, wie viele Elemente eines bestimmten Typs sich in einer Datei befinden, z.B. `10` Elemente vom Typ `unsigned int`, kann dies auch direkt mit `fread`, d.h. ohne Schleife, erledigt werden.

In einer Datei mit Ninja Bytecode ist bekannt wie viele Instruktionen enthalten sind!

```c
int item_count = 10; // we expect to have 10 items in the file!
unsigned int items[item_count]; // make space

/* we try to read 10 items at once!*/
read_objects = fread(&inst, sizeof(unsigned int), item_count, fp);
if(read_objects != item_count) {
    printf("ERROR: Could only read [%lu] of [%d] items.\n", read_objects, item_count);
    exit(1);
}
```

BEISPIEL: LESEN VON ELEMENTEN OHNE SCHLEIFE

Der u.a. Beispielcode zeigt, wie man ohne Schleife mittels `fread` mehrere Elemente auslesen kann.

	In der Datei `test.bin` sind nur 9 `unsigned int`-Elemente (36 Bytes) vorhanden. Somit löse der Code den Fehlerfall für gelesene Objekte aus!

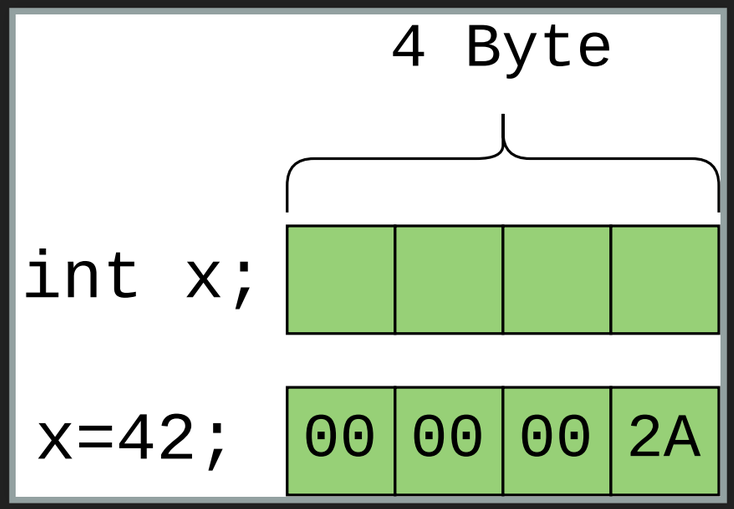
```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    FILE * fp;
    size_t read_objects;
    int item_count = 10; // we expect to have 10 items in the file!
    unsigned int items[item_count]; // make space
    if ((fp = fopen("./test.bin", "r"))==NULL) {
        perror("fopen");
        exit(1);
    }
    /* we try to read 10 items at once!*/
    read_objects = fread(items, sizeof(unsigned int), item_count, fp);
    if(read_objects != item_count) {
            printf("ERROR: Could only read [%lu] of [%d] items.\n", read_objects, item_count);
                exit(1);
    }
    return 0;
}
```

EINSCHUB: BASISWISSEN ZEIGER

Ohne ein Verständnis was **Zeiger** (Pointer) sind und wie man sie verwendet, ist eine sinnvolle Programmierung in der Programmiersprache `C` nicht möglich.

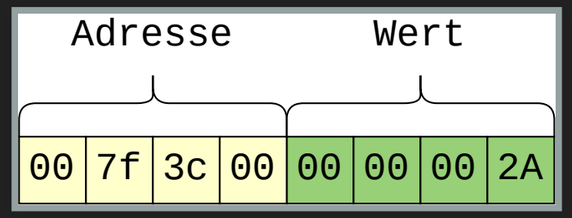
Hierzu ist es unabdingbar grundsätzlich zu verstehen, wie die Speicherverwaltung auf Computern funktioniert.

- Die Definition einer Variablen (z.B. `int x;`) reserviert im Speicher eine bestimmte Anzahl an Bytes
  - Typischerweise sind das auf der X86_64 Architektur 4 Byte für einen Integer
- Eine Zuweisung eines Wertes an eine Variable (z.B. `x=42;`) speichert nun den Wert `42` (`0x2A`) im reservierten Speicher ab



EINSCHUB: ZEIGER - SPEICHERADRESSEN

Die Variable `x` liegt nun *irgendwo* im Speicher. Nehmen wir einmal an die **Speicheradresse** von `x` wäre `0x007f3c00`



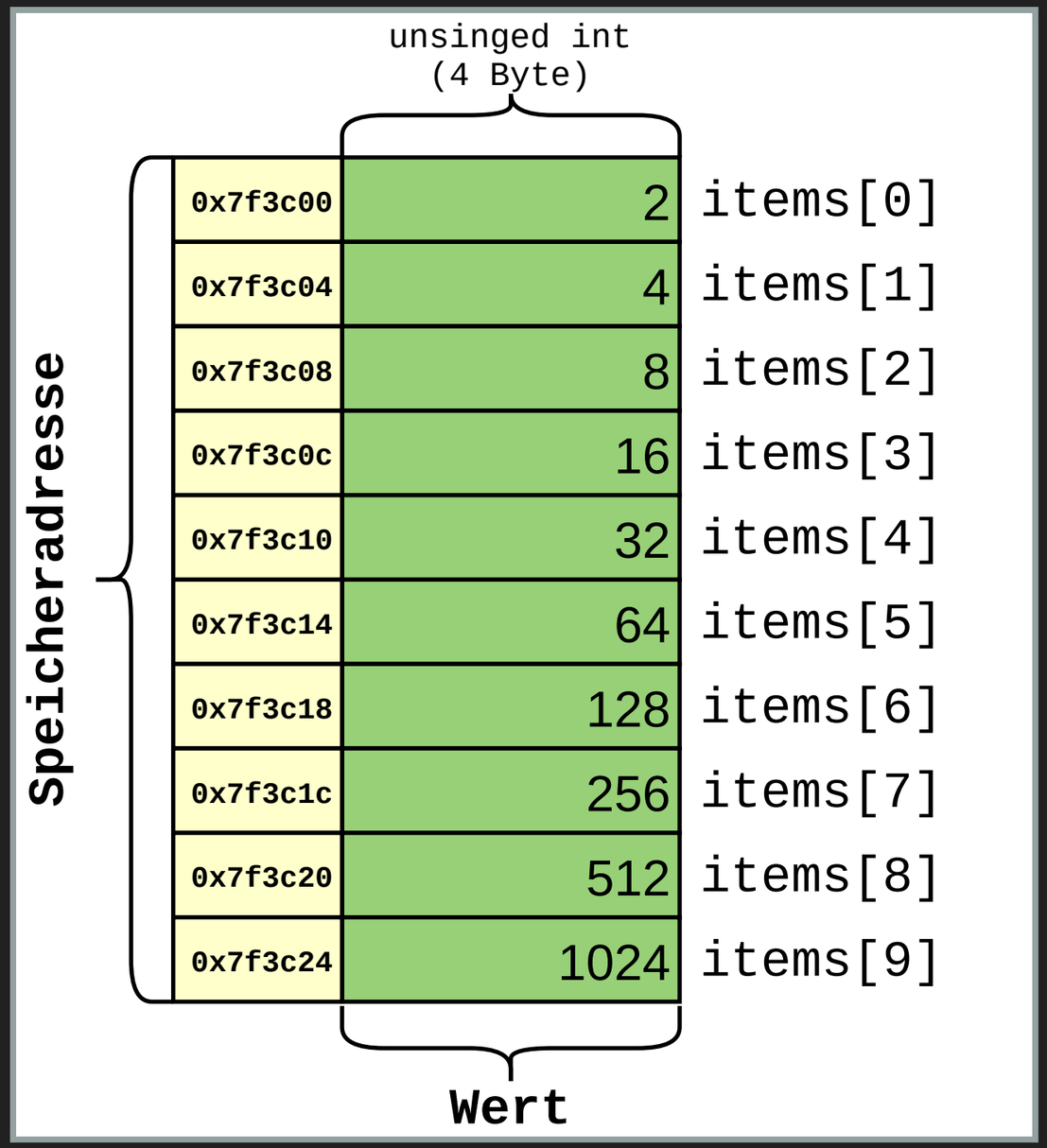
- Wenn wir auf die Speicheradresse von `x` zugreifen möchten, verwenden wir das *kaufmännische UND-Zeichen* `&` - Dies wird auch **Referenzieren** (Pointer-Konstruktion) genannt!
  - Es wird gelesen als "Adresse von" → `&x` = Adresse von `x`
- Eine Ausgabe der Adresse und des Wertes könnte nun z.B. erfolgen mit:

```
printf("Adresse: %p, Wert: %d (0x%x)", &x, x, x);

AUSGABE:

Adresse: 0x007f3c00, Wert: 42 (0x2a)
```

- Der **Typ** von `&x` ist `int *` — ein Zeiger auf einen Integer.
- Der **Wert** von `&x` ist die Adresse von `x`.

EINSCHUB: ZEIGER AUF ARRAYFELDER

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

int main(int argc, char *argv[])
{
    int offset = 0x7f3c00;
    unsigned int items[]={2,4,8,16,32,64,128, 256, 512, 1024};
    for (int i=0;i<10;i++){
        int address=offset+(i*sizeof(unsigned int));
        printf("&item[%d] (0x%06x) -> |%4d| item[%d]\n",
                i, address, items[i], i);
    }
    return 0;
}
```



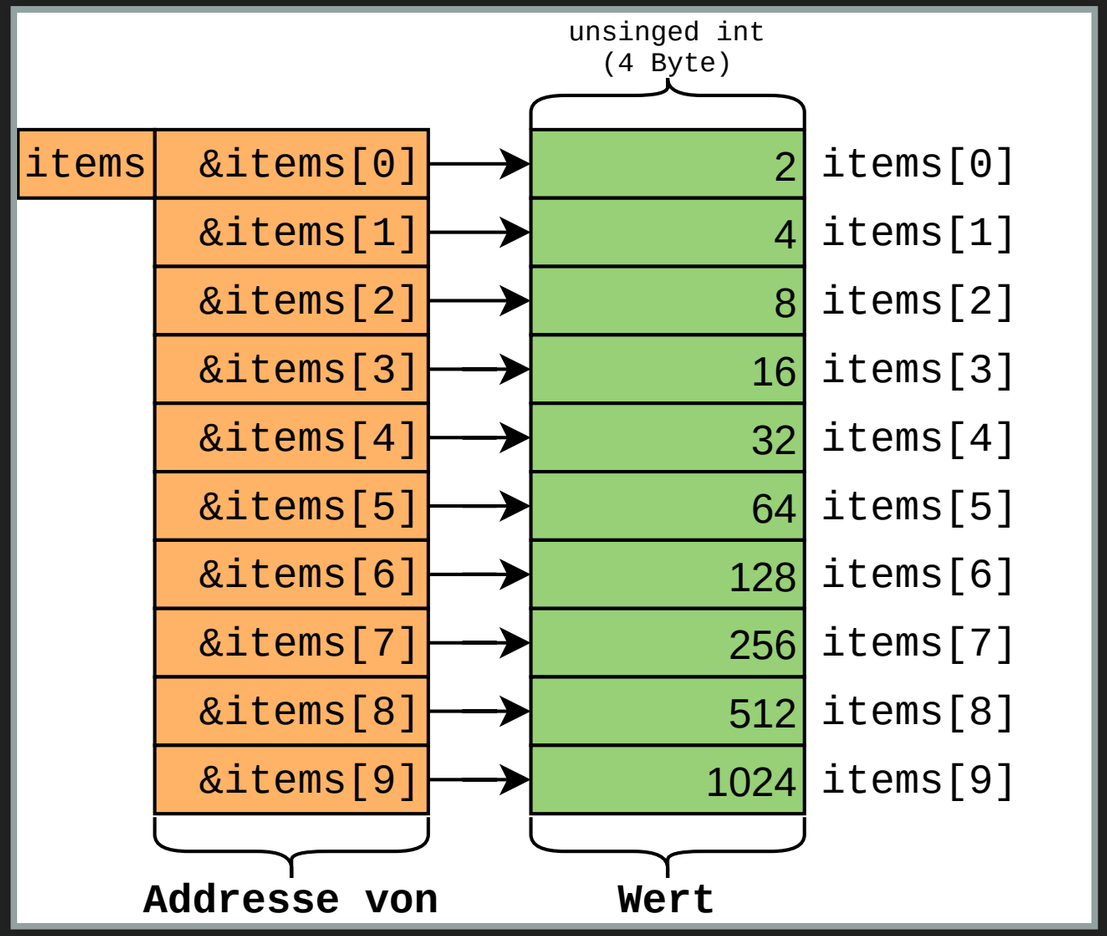
`address` ist hier zum besseren Verständnis künstlich erzeugt worden!

```
$ gcc -Wall pointer1.c -o pointer1 && ./pointer1
&item[0] (0x7f3c00) -> |   2| item[0]
&item[1] (0x7f3c04) -> |   4| item[1]
&item[2] (0x7f3c08) -> |   8| item[2]
&item[3] (0x7f3c0c) -> |  16| item[3]
&item[4] (0x7f3c10) -> |  32| item[4]
&item[5] (0x7f3c14) -> |  64| item[5]
&item[6] (0x7f3c18) -> | 128| item[6]
&item[7] (0x7f3c1c) -> | 256| item[7]
&item[8] (0x7f3c20) -> | 512| item[8]
&item[9] (0x7f3c24) -> |1024| item[9]
```

EINSCHUB: ZEIGER AUF ARRAYFELDER

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

int main(int argc, char *argv[])
{
    int offset = 0x7f3c00;
    unsigned int items[]={2,4,8,16,32,64,128, 256, 512, 1024};
    for (int i=0;i<10;i++){
        int address=offset+(i*sizeof(unsigned int));
        printf("&item[%d] (0x%06x) -> |%4d| item[%d]\n",
                i, address, items[i], i);
    }
    return 0;
}
```

```
$ gcc -Wall pointer1.c -o pointer1 && ./pointer1
&item[0] (0x7f3c00) -> |   2| item[0]
&item[1] (0x7f3c04) -> |   4| item[1]
&item[2] (0x7f3c08) -> |   8| item[2]
&item[3] (0x7f3c0c) -> |  16| item[3]
&item[4] (0x7f3c10) -> |  32| item[4]
&item[5] (0x7f3c14) -> |  64| item[5]
&item[6] (0x7f3c18) -> | 128| item[6]
&item[7] (0x7f3c1c) -> | 256| item[7]
&item[8] (0x7f3c20) -> | 512| item[8]
&item[9] (0x7f3c24) -> |1024| item[9]
```



`items` ohne Angabe eines Indices eines Feldes ist äquivalent zu `&items[0]`!

EINSCHUB: ZEIGERTYPEN

- Variablen können selbst vom Typ `Zeiger auf Typ` sein
  - `int *y` → Typ: Zeiger auf `int`
  - `char *c` → Typ: Zeiger auf `char` (*auch als String bekannt!*)
  - `long *l` → Typ: Zeiger auf `long`
  - `char **v` → Typ: Zeiger auf Zeiger auf `char` (vgl. `char * argv[]`)
  - `void *p` → Typ: Zeiger auf unbestimmten Typ

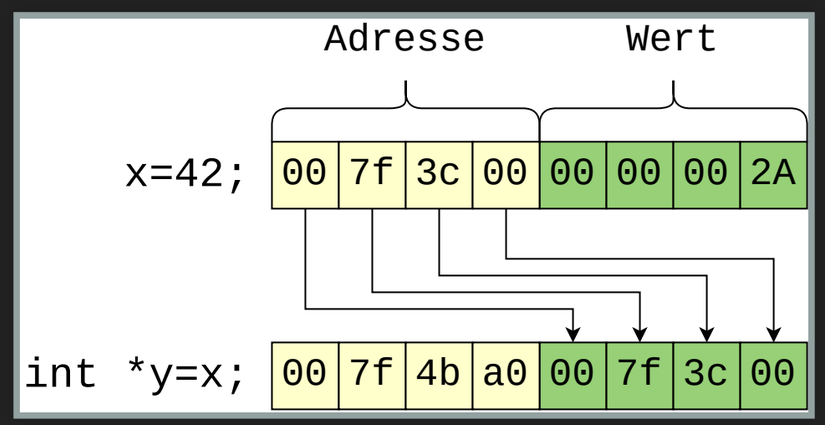
Der Wert vom Typ `Zeiger auf Typ` ist **immer** eine Adresse!

- Die Anzahl an Bytes (Größe), die von einem Zeiger im Speicher belegt werden, ist Architekturabhängig:
  - **X86** (IA-32): 4 Byte (32-Bit Speicheradressen)
  - **X86_64** (AMD64): 8 Byte (64-Bit Speicheradressen)
    - Auf X86_64 werden jedoch effektiv nur 6 Byte genutzt. Die 2 MSB haben den Wert `0x00`!

EINSCHUB: ZEIGERTYPEN UND DEREFERENZIERUNG

- Wenn wir im u.a. Beispiel sagen:
  - `y` *zeigt auf* `x` oder
  - `y` *ist ein Zeiger auf* `x`
- Dann meinen wir eigentlich, dass `y` der Wert der **Speicheradresse** von `x` zugewiesen wurde.

```c
int x = 42; // Wert = 42
int *y=&x;  // Wert = Adresse von x
printf("x=%d <-> *y=%d | &x=%p <-> y=%p\n", x, *y, &x, y);

AUSGABE:
x=42 <-> *y=42 | &x=0x007f3c00 <-> y=0x007f3c00
```



Wenn wir nun mittels `y` auf den **Wert** von `x` zugreifen wollen, dann müssen wir eine sog. **Dereferenzierung** mit dem `*`-Operator durchführen.

EINSCHUB: ZEIGER AUF ARRAYS

Eigentlich verwenden wir schon die ganze Zeit Zeiger — nämlich immer wenn wir ein Array angelegt haben, werden intern Zeiger verwendet.

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int array[]={1,2,3};
    printf("%d = %d = %d \n", array[1], *(array+1), *(&array[0]+1));
    return 0;
}

AUSGABE:
2 = 2 = 2
```

- Der Ausdruck: `array[1]` ist äquivalent zu `*(array+1)` bzw. zu `*(&array[0]+1)`

ZEIGER: BEISPIEL MIT VERSCHIEDENEN ZEIGERN

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

int main(int argc, char *argv[])
{
    unsigned long long_items[]={2,4,8,16,32,64,128, 256, 512, 1024};
    unsigned int int_items[]={2,4,8,16,32,64,128, 256, 512, 1024};
    char char_items[]={'A','B','C','D','E','F','G','H','I','J'};

    unsigned long *li_ptr = &long_items[0];
    unsigned int *ui_ptr = int_items;
    char *ch_ptr=&char_items[0];

    for (int i=0;i<10;i++){
        printf("&long_item[%d] (%p) = li_ptr+%d (%p) -> |%4lu| \n",
               i, &long_items[i], i, li_ptr+i, *(li_ptr+i));
    }
    puts("");
    for (int i=0;i<10;i++){
        printf("&int_item[%d]  (%p) = ui_ptr+%d (%p) -> |%4d| \n",
               i, &int_items[i], i, ui_ptr+i, *(ui_ptr+i));
    }
    puts("");
    for (int i=0;i<10;i++){
        printf("&char_item[%d] (%p) = ch_ptr+%d (%p) -> |%4c| \n",
               i, &char_items[i], i, ch_ptr+i, *(ch_ptr+i));
    }
    return 0;
}
```

```
&long_item[0] (0x7ffe81763030) = li_ptr+0 (0x7ffe81763030) -> |   2|
&long_item[1] (0x7ffe81763038) = li_ptr+1 (0x7ffe81763038) -> |   4|
&long_item[2] (0x7ffe81763040) = li_ptr+2 (0x7ffe81763040) -> |   8|
&long_item[3] (0x7ffe81763048) = li_ptr+3 (0x7ffe81763048) -> |  16|
&long_item[4] (0x7ffe81763050) = li_ptr+4 (0x7ffe81763050) -> |  32|
&long_item[5] (0x7ffe81763058) = li_ptr+5 (0x7ffe81763058) -> |  64|   //LONG=+8
&long_item[6] (0x7ffe81763060) = li_ptr+6 (0x7ffe81763060) -> | 128|
&long_item[7] (0x7ffe81763068) = li_ptr+7 (0x7ffe81763068) -> | 256|
&long_item[8] (0x7ffe81763070) = li_ptr+8 (0x7ffe81763070) -> | 512|
&long_item[9] (0x7ffe81763078) = li_ptr+9 (0x7ffe81763078) -> |1024|

&int_item[0]  (0x7ffe81763000) = ui_ptr+0 (0x7ffe81763000) -> |   2|
&int_item[1]  (0x7ffe81763004) = ui_ptr+1 (0x7ffe81763004) -> |   4|
&int_item[2]  (0x7ffe81763008) = ui_ptr+2 (0x7ffe81763008) -> |   8|
&int_item[3]  (0x7ffe8176300c) = ui_ptr+3 (0x7ffe8176300c) -> |  16|
&int_item[4]  (0x7ffe81763010) = ui_ptr+4 (0x7ffe81763010) -> |  32|
&int_item[5]  (0x7ffe81763014) = ui_ptr+5 (0x7ffe81763014) -> |  64|  //INT=+4
&int_item[6]  (0x7ffe81763018) = ui_ptr+6 (0x7ffe81763018) -> | 128|
&int_item[7]  (0x7ffe8176301c) = ui_ptr+7 (0x7ffe8176301c) -> | 256|
&int_item[8]  (0x7ffe81763020) = ui_ptr+8 (0x7ffe81763020) -> | 512|
&int_item[9]  (0x7ffe81763024) = ui_ptr+9 (0x7ffe81763024) -> |1024|

&char_item[0] (0x7ffe8176308e) = ch_ptr+0 (0x7ffe8176308e) -> |   A|
&char_item[1] (0x7ffe8176308f) = ch_ptr+1 (0x7ffe8176308f) -> |   B|
&char_item[2] (0x7ffe81763090) = ch_ptr+2 (0x7ffe81763090) -> |   C|
&char_item[3] (0x7ffe81763091) = ch_ptr+3 (0x7ffe81763091) -> |   D|
&char_item[4] (0x7ffe81763092) = ch_ptr+4 (0x7ffe81763092) -> |   E| //CHAR=+1
&char_item[5] (0x7ffe81763093) = ch_ptr+5 (0x7ffe81763093) -> |   F|
&char_item[6] (0x7ffe81763094) = ch_ptr+6 (0x7ffe81763094) -> |   G|
&char_item[7] (0x7ffe81763095) = ch_ptr+7 (0x7ffe81763095) -> |   H|
&char_item[8] (0x7ffe81763096) = ch_ptr+8 (0x7ffe81763096) -> |   I|
&char_item[9] (0x7ffe81763097) = ch_ptr+9 (0x7ffe81763097) -> |   J|
```

Beachten Sie insbesondere, wie sich die Adressen bei den unterschiedlichen Datentypen erhöhen. Obwohl, *vermeintlich immer*, die gleiche Zahl `i` auf den jeweiligen Zeiger addiert wird!
`long` → +8 Bytes, `int` → +4 Bytes, `char` → +1 Byte

ZEIGER: `void *ptr`

Der Zeiger auf einen unbestimmten Typ `void *ptr` (sog. *void pointer*) wird sehr oft verwendet. Insbesondere lassen sich mit dieser Art Zeiger auch Funktionen an andere Funktionen übergeben. Generell kann hiermit also auf **beliebige** Typen an Daten gezeigt werden.

Man kann jedoch einen void pointer nicht direkt dereferenzieren. Hierzu muss zuerst auf einen bestimmten Datentyp umgewandelt (*gecastet*) werden. Dies ist notwendig, damit der Compiler weiß, auf welche Adressen zugegriffen werden soll, da sich, je nach Datentyp, die Größe der belegten Speicherbereiche (für die hinterlegten Werte) verändert. Das ist insbesondere bei der sog. Zeiger-Arithmetik, also Berechnungen mit Zeigern, wichtig!

ZEIGER: BEISPIEL `void *ptr`

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned int x = 42;
    long y = 43;
    int array[]={5,4,3,2,1};
    void *ptr;
    ptr=&x; // now pointing to usigned int
    //printf("%p -> %d\n", ptr, *ptr); // (compilation error: invalid use of void expression)
    printf("%p -> %d\n", ptr, *(int *)ptr);
    ptr=&y; // now pointing to long
    printf("%p -> %lu\n", ptr, *(long *)ptr);
    ptr=array; // now pointing to an array of int's
    printf("%p -> %2d\n", ptr, *(int *)ptr);
    ptr=((int *) ptr)+2; // no ptr++ - Arithmetik on void pointers is not allowed! ptr -> &array[2]
    printf("%p -> %2d\n", ((int *)ptr), *((int *)ptr)); // we modified ptr, so watch closely the output
    printf("%p -> %2d\n", ((int *)ptr+2), *((int *)ptr+2)); // Remember ptr points to &array[2]
    return 0;
}
```

```
$ gcc -Wall pointer5.c -o pointer5 && ./pointer5
0x7ffe88103e7c -> 42
0x7ffe88103e80 -> 43
0x7ffe88103e90 ->  5
0x7ffe88103e98 ->  3
0x7ffe88103ea0 ->  1
```

Achten Sie beim Dereferenzieren, insbesondere bei Arrays und später bei Strukturen (`struct`), unbedingt auf eine korrekte Klammerung!

SPEICHERANFORDERUNG UND FREIGABE

Lokale Variablen befinden sich auf bei `C` auf dem Laufzeitstack. Beim Aufruf der Funktion `f()` wird auf dem aktuellen Stack Platz für alle lokalen Variablen geschaffen. Die Variablen können dann wie gewohnt verwendet werden.

```c
int f(void) {
    int a; (1)
    int b; (1)
    a=33;
    b=a+1;
    return a; (2)
}
```

(1) Befinden sich auf dem `C`-Laufzeitstack für `f()`

(2) Funktionsende

Nach Funktionsende sind sowohl die Speicherplätze der Variablen, als auch die zugewiesenen oder berechneten Werte verloren (bzw. **nicht** mehr **gültig**!). Der Speicher wird hierbei automatisch angefordert und freigegeben!

	Es geht hier um den `C`-Laufzeitstack, nicht um die NinjaVM (obwohl es hier ähnlich sein wird!)

SPEICHERANFORDERUNG UND FREIGABE

Lokale Variablen befinden sich auch bei `C` auf dem Laufzeitstack. Beim Aufruf der Funktion `f()` wird auf dem aktuellen Stack Platz für alle lokalen Variablen geschaffen. Die Variablen können dann wie gewohnt verwendet werden.

```c
int* f2(void) {
    int i = 25; (1)
    return &i;  (2)
}
```

1. Befinden sich auf dem `C`-Laufzeitstack für `f2()`
2. Funktionsende: Rückgabewert `&i` zeigt auf (semantisch) ungültigen Speicher.

⚠	Obwohl die Funktion `f2()` ohne Fehler kompiliert, ist das Ergebnis sicherlich nicht das, was Sie erwarten würden! Aktuelle Compiler setzen den Rückgabewert in Register `rax` auf `0x00` (`mov eax,0x0`) → es wird also hierbei ein Zeiger auf `NULL` zurückgegeben, der, sobald er dereferenziert wird, einen Laufzeitfehler auslöst.

SPEICHERANFORDERUNG UND FREIGABE: BEISPIEL

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

int* f2(void) {
    int i = 25;
    printf("&i= %p\n", &i);
    return &i;
}

int main(int argc, char *argv[]) {
    int *j = f2();
    printf("j= %p\n", j);
    printf("j= %d\n", *j);
    return 0;
}
```

```
$ gcc -Wall -g malloc1.c -o malloc1 && ./malloc1
malloc1.c: In function ‘f2’:
malloc1.c:7:12: warning: function returns address of local variable [-Wreturn-local-addr]
    7 |     return &i;
      |            ^~
&i= 0x7ffc812ee894
j= (nil)
Segmentation fault (core dumped)
```

SPEICHERANFORDERUNG UND FREIGABE: MALLOC

Dynamische Speicheranforderung: Der **Heap**-Speicherbereich ermöglicht den Zugriff auf Variablen (oder generell Daten) über Funktionsaufrufe hinweg. D.h., auf diesen Speicherbereich können alle Funktionen eines laufenden Programms **uneingeschränkt** zugreifen.

Uneingeschränkt bedeutet hier, dass bei konkurrierenden Zugriffen auf Speicherbereiche im Heap, Synchronisationsmechanismen (wie z.B. *Semaphoren* oder `Mutexe`) verwendet werden müssen. Im KsP haben wir aber glücklicherweise keine konkurrierenden Zugriffe auf Speicherbereiche! :) → Das ist aber Thema in Betriebssysteme!

SPEICHERANFORDERUNG: MALLOC

- Mit der Funktion `void *malloc(size_t size);` kann man auf dem Heap Speicher anfordern.
  - **Parameter**: Anzahl an Bytes die reserviert werden sollen.
  - **Rückgabewert**: Als Rückgabe erhält man einen Zeiger auf den reservierten Speicherbereich (d.h. eine gültige Adresse, die je nach Deklaration einem spezifischen Typ zugeordnet wird)! Man muss hierbei also nicht explizit *casten*.
    - Im Fehlerfall wird als Rückgabewert `NULL` zurückgegeben.

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

void f1(int *v) {
    *v+=100;
    printf("--(%4s): v (@%p) = %d\n",__func__, v, *v);
}

int* f2(void) {
    int *i;
    if ((i=malloc(sizeof(int))) == NULL){
        perror("malloc");
        exit(1);
    }
    *i=5;
    printf("--(%4s): i (@%p) = %d\n",__func__, i, *i);
    return i;
}

int main(int argc, char *argv[]) {
    int *j = f2();
    f1(j);
    printf("--(%4s): j (@%p) = %d\n",__func__, j, *j);
    free(j);
    return 0;
}
```

```
$ gcc -Wall -g malloc2.c -o malloc2 && ./malloc2
--(  f2): i (@0x5558951aa2a0) = 5
--(  f1): v (@0x5558951aa2a0) = 105
--(main): j (@0x5558951aa2a0) = 105
```

`void *function()` gibt einen **Zeiger** auf einen **unbestimmten Typ** zurück.

`void function()` gibt **nichts** zurück!

SPEICHERFREIGABE: FREE

Wenn man Speicherplatz auf dem Heap nicht länger benötigt, muss man ihn in `C` wieder explizit freigeben. `C` hat keine sog. *Garbage Collection*, d.h. der Entwickler ist selbstständig für die Freigabe von Speicher verantwortlich.

- **Speicherfreigabe**: `void free(void *ptr);`
- **Parameter**: Speicheradresse die freigegeben werden soll.
  - Die Adresse muss eine Adresse sein, die vorher von einem Aufruf von `malloc` stammt!

	`free` gibt den gesamten Speicherbereich frei, der mit der übergebenen Adresse assoziiert ist. D.h. wenn ein Speicherbereich von 1024 Bytes reserviert wurde (`char *s=malloc(1024)`), werden auch 1024 Bytes freigegeben (`free(s)`).

`malloc` und `free` sind komplexe Funktionen, die sowohl Speicher reservieren, freigeben und auch zugehörige Metadaten (z.B. die Größe des Speicherbereichs) verwalten.

SPEICHERFREIGABE: HÄUFIGE FEHLER

Es ist nicht unüblich, dass vergessen wird, reservierten Speicher wieder freizugeben oder auf bereits freigegebenen Speicher zuzugreifen (Sicherheitslücke: *use-after-free*).

Beispiel: Zugriff auf bereits freigegebenen Speicher.

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

int main(int argc, char *argv[]) {
    int i=100;
    char *string;
    if ((string=malloc(1024)) == NULL){
        perror("malloc");
        exit(1);
    }
    sprintf(string, "function: %s, value of i = %d",
            __func__,i);
    printf("|%s|\n", string);
    free(string);
    printf("|%s|\n", string);
    puts("");
    return 0;
}
```

```
$ gcc -Wall -g free1.c -o free1
$ ./free
|function: main, value of i = 100|
||D]|

$ ./free
|function: main, value of i = 100|
|K)Y|

$ ./free
|function: main, value of i = 100|
|D0V|

$ ./free
|function: main, value of i = 100|
|N[\|
```

Nach der Speicherfreigabe durch `free()` enthält die Ausgabe **beliebige** Daten! Die kann uns später Probleme machen!

SPEICHERFREIGABE: HÄUFIGE FEHLER

Beispiel: Speicher wird nicht freigegeben

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

int main(int argc, char *argv[]) {
    int i=100;
    char *string;
    if ((string=malloc(1024)) == NULL){
        perror("malloc");
        exit(1);
    }
    sprintf(string, "function: %s, value of i = %d",
            __func__,i);
    printf("|%s|\n", string);
    return 0;
}
```

```
gcc -Wall -g free2.c -o free2  && ./free2
|function: main, value of i = 100|
```

Keine Probleme erkannt! Aber unser Speicher kann u.U. irgendwann voll werden!

Nicht mehr benutzter Speicher sollte immer freigegeben werden, sobald er nicht mehr benötigt wird. Fehler können hierbei mit dem Tool `valgrind` erkannt werden.

SPEICHERFREIGABE: HÄUFIGE FEHLER (VALGRIND)

Mit `valgrind` kann ein Programm auf Speicherfehler überprüft werden. In diesem Beispiel erkennt `valgrind`, dass reservierter Speicher nicht freigeben wurde. `valgrind` kann auch andere Fehler, wie z.B. Zugriff auf *fehlerhafte* Speicherbereiche (z.B. zu kleine), erkennen.

```
$ gcc -Wall -g free2.c -o free2
$ valgrind --leak-check=full ./free2
...
|function: main, value of i = 100|
==1400873==
==1400873== HEAP SUMMARY:
==1400873==     in use at exit: 1,024 bytes in 1 blocks (1)
==1400873==   total heap usage: 2 allocs, 1 frees, 2,048 bytes allocated
==1400873==
==1400873== 1,024 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1400873==    at 0x483A77F: malloc (vg_replace_malloc.c:307)
==1400873==    by 0x109198: main (free2.c:7) (2)
==1400873==
==1400873== LEAK SUMMARY:
==1400873==    definitely lost: 1,024 bytes in 1 blocks (1)
==1400873==    indirectly lost: 0 bytes in 0 blocks
==1400873==      possibly lost: 0 bytes in 0 blocks
==1400873==    still reachable: 0 bytes in 0 blocks
==1400873==         suppressed: 0 bytes in 0 blocks
==1400873==
==1400873== For lists of detected and suppressed errors, rerun with: -s
==1400873== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0
```

1. Speicherfehler erkannt
2. Hier wurde der Speicher reserviert

SPEICHERANFORDERUNG UND FREIGABE: BEISPIEL 1

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

#define MAX 5
int main(int argc, char *argv[]) {
    unsigned long *li_ptr = malloc(MAX*sizeof(unsigned long)); 1
    unsigned int *ui_ptr = malloc(MAX*sizeof(unsigned int)); 1
    char *ch_ptr=malloc(MAX*sizeof(char)); 1
    if (li_ptr == NULL || ui_ptr == NULL || ch_ptr == NULL){
        fprintf(stderr, "ERROR during a malloc() call\n");
        exit(1);
    }
    for (int i=0;i<MAX;i++){ 2
        li_ptr[i]=pow(2,i+1); 2
        ui_ptr[i]=pow(2,i+1); 2
        ch_ptr[i]='A'+i; 2
    }
    for (int i=0;i<MAX;i++){
        printf("&li_ptr[%d] (%p) = li_ptr+%d (%p) -> |%4lu| \n",
                i, &li_ptr[i], i, li_ptr+i, *(li_ptr+i));
    }
    puts("");
    for (int i=0;i<MAX;i++){
        printf("&ui_ptr[%d] (%p) = ui_ptr+%d (%p) -> |%4d| \n",
                i, &ui_ptr[i], i, ui_ptr+i, ui_ptr[i]);
    }
    puts("");
    for (int i=0;i<MAX;i++){
        printf("&ch_ptr[%d] (%p) = ch_ptr+%d (%p) -> |%4c| \n",
                i, &ch_ptr[i], i, ch_ptr+i, *(ch_ptr+i));
    }
    free(li_ptr); 3
    free(ui_ptr); 3
    free(ch_ptr); 3
    return 0;
}
```

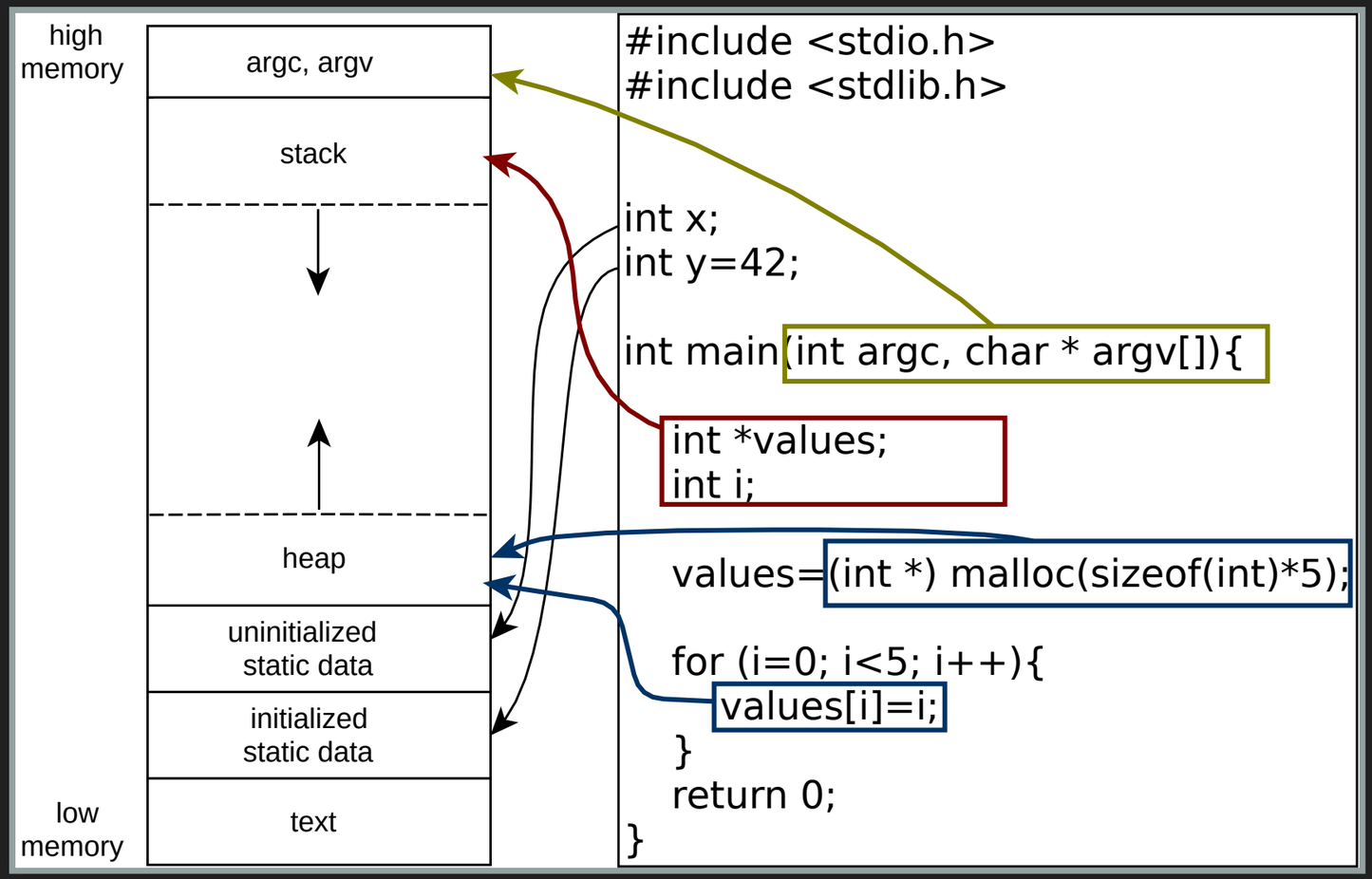
```
&li_ptr[0] (0x4b8f040) = li_ptr+0 (0x4b8f040) -> |   2|
&li_ptr[1] (0x4b8f048) = li_ptr+1 (0x4b8f048) -> |   4|
&li_ptr[2] (0x4b8f050) = li_ptr+2 (0x4b8f050) -> |   8|
&li_ptr[3] (0x4b8f058) = li_ptr+3 (0x4b8f058) -> |  16|
&li_ptr[4] (0x4b8f060) = li_ptr+4 (0x4b8f060) -> |  32|

&ui_ptr[0] (0x4b8f0b0) = ui_ptr+0 (0x4b8f0b0) -> |   2|
&ui_ptr[1] (0x4b8f0b4) = ui_ptr+1 (0x4b8f0b4) -> |   4|
&ui_ptr[2] (0x4b8f0b8) = ui_ptr+2 (0x4b8f0b8) -> |   8|
&ui_ptr[3] (0x4b8f0bc) = ui_ptr+3 (0x4b8f0bc) -> |  16|
&ui_ptr[4] (0x4b8f0c0) = ui_ptr+4 (0x4b8f0c0) -> |  32|

&ch_ptr[0] (0x4b8f110) = ch_ptr+0 (0x4b8f110) -> |   A|
&ch_ptr[1] (0x4b8f111) = ch_ptr+1 (0x4b8f111) -> |   B|
&ch_ptr[2] (0x4b8f112) = ch_ptr+2 (0x4b8f112) -> |   C|
&ch_ptr[3] (0x4b8f113) = ch_ptr+3 (0x4b8f113) -> |   D|
&ch_ptr[4] (0x4b8f114) = ch_ptr+4 (0x4b8f114) -> |   E|
```

1. Speicheranforderung auf dem Heap mit `malloc()`
2. Eine Initialisierung über `{2,4,8,...}` ist nicht mehr möglich!
3. Speicherfreigabe mit `free()`

Beachten Sie wieder die Werte der Speicheradressen

ÜBERSICHT PROGRAMMSPEICHER (`C`)

Übersicht wie ein Programm zur Laufzeit im Speicher abgebildet ist und Zuordnung von `C`-Programmbestandteilen in den jeweiligen zuständigen Speicherbereich.



KURZZUSAMMENFASSUNG:

Es stehen nun alle Werkzeuge zur Verfügung, um ein Ninja Programm, das in Ninja Bytecode in einer Datei (`xxx.bin`) vorhanden ist, in die Ninja VM zu laden und den Speicher der Ninja VM mit dynamischer Speicherverwaltung zu implementieren.

- **Dateioperationen**: Öffnen, Schließen und Lesen aus einer Datei.
- **Zeiger**: Was sind Speicheradressen, was sind Zeiger und wie werden sie verwendet.
- **Speichermanagement**: Speicheranforderung und Freigabe mittels `malloc()` und `free()`.