Technische Informatik 3 – Embedded Systems

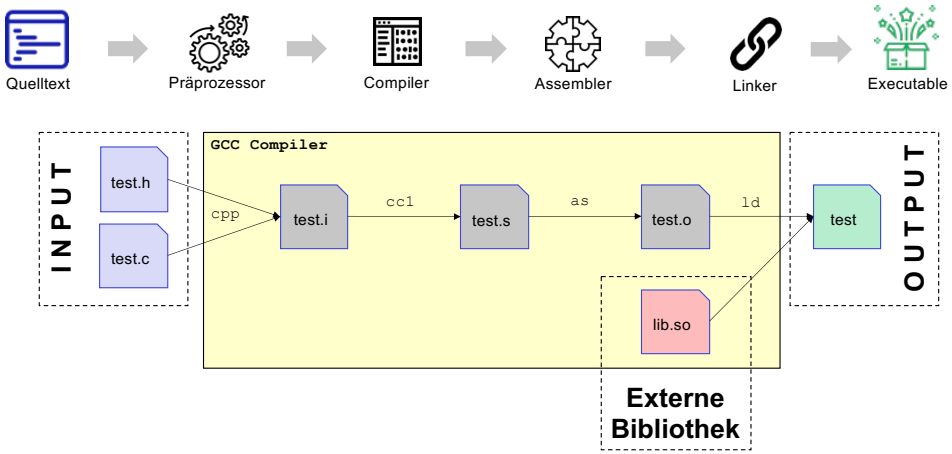
Kapitel 4: Strukturen

Prof. Dr. Benjamin Kormann
Fakultät für Elektro- und Informationstechnik
15.05.2023

Exkursion zum Präprozessor
Rekapitulation

Der Präprozessor in C

- Während des Kompiliervorgangs verarbeitet zuerst der Präprozessor die Quelltextdateien
- Dieser wertet im wesentlichen die `#`-Direktiven aus
- Häufigste Verwendung neben `#include` sind `#define`



Grobe Funktionsweise an einem Beispiel

```
1  #include <stdio.h>
2  #define PI 3.14
3
4  int main()
5  {
6      double radius = 2.0;
7      double area = radius * radius * PI;
8      printf("Area: %lf\n", area);
9      return 0;
10 }
```

Präprozessor

```
726 int main()
727 {
728     double radius = 2.0;
729     double area = radius * radius * 3.14;
730     printf("Area: %lf\n", area);
731     return 0;
732 }
```

#-Anweisungen aufgelöst

Simples Suchen & Ersetzen

Exkursion zum Präprozessor
Gefahren von `#define`

Definition von Funktionen mit Hilfe von `#define`

Fall 1

```c
#include <stdio.h>
#define MAX(a, b) (a > b) ? a : b

int main()
{
    int x = 5;
    int y = 6;
    int max = MAX(x, y);
    printf("MAX: %d\n", max);
    return 0;
}
```

```c
int main()
{
    int x = 5;
    int y = 6;
    int max = (x > y) ? x : y;
    printf("MAX: %d\n", max);
    return 0;
}
```

```
MAX: 6
```

Fall 2

```c
#include <stdio.h>
#define MAX(a, b) (a > b) ? a : b

int main()
{
    int x = 5;
    int y = 6;
    int max = MAX(x, ++y);
    printf("MAX: %d\n", max);
    return 0;
}
```

Präinkrement

```c
int main()
{
    int x = 5;
    int y = 6;
    int max = (x > ++y) ? x : ++y;
    printf("MAX: %d\n", max);
    return 0;
}
```

Doppeltes Präinkrement

```
MAX: 8
```

Strukturen in der Programmiersprache C

Was sind Strukturen?

- Selbstdefinierte, zusammengesetzte Datentypen
- Sie beinhalten eine fest definierte Anzahl von Komponenten, die von unterschiedlichen Typen sein können

Einsatzmöglichkeiten von Strukturen

- Mehrere Daten können somit in einer einzigen Struktur (über eine Variable) angesprochen werden
- Definition von problembezogenen Datenstrukturen
- Einzelne Komponenten der Datenstruktur können über den Namen angesprochen werden, nicht wie über einen Index wie bei Arrays

Anlegen einer Struktur für komplexe Zahlen
(; ist wichtig für Definition)

```c
#include <stdio.h>

int main()
{
    // Definition einer Struktur
    struct Complex
    {
        double re;
        double im;
    };

    // Anlegen einer Instanz der Struktur
    struct Complex z;
    // Zugriff auf Elemente der Struktur
    z.re = 1.1;
    z.im = 2.2;

    return 0;
}
```

Verwenden der Struktur

Strukturen in der Programmiersprache C
Beispiele

Definition

```c
// Definition der Struktur date
struct date
{
    int day;
    int month;
    int year;
    char name[4];
};
```

- Enthält 4 Elemente
  - 3 `int`
  - 4 `char` als Array

Definition und Verwendung

```c
// Definition der Struktur date
// Anlegen von Instanzen mit Namen:
//  today und birthday
struct date
{
    int day;
    int month;
    int year;
    char name[4];
} today, birthday;
```

- Enthält 4 Elemente
  - 3 `int`
  - 4 `char` als Array
- Instanzen `today` und `birthday` der Struktur `date` werden erzeugt

Definition und Verwendung 2

```c
// Definition der Struktur date
// Anlegen einer Instanz mit Namen:
//  start
struct point
{
    double xpos;
    double ypos;
    int value;
    char label;
} start;

// Anlegen weiterer Instanzen p1 und p2
struct point p1, p2;
```

- Enthält 4 Elemente
  - 2 `double`, `1 int`
  - 1 `char`
- Instanz `start` wird mit der Definition angelegt
- Instanzen `p1` und `p2` erzeugt

Strukturen in der Programmiersprache C
Verwenden einer Struktur mit Hilfe von Zeigern

Zugriff auf die Elemente einer Struktur

- Bei einer Instanz einer Struktur kann auf die einzelnen Elemente mit Hilfe des . Operators zugegriffen werden
- Wird ein Zeiger auf einer Struktur verwendet, so muss mit Hilfe der -> Operators auf die einzelnen Elemente zugegriffen werden

Wofür wird Zeiger auf Struktur benötigt?

- Übergabe an Funktionen gemäß call-by-reference
- Dynamische Speicherverwaltung

```c
#include <stdio.h>

int main()
{
    // Definition einer Struktur
    struct Complex
    {
        double re;
        double im;
    };

    // Anlegen einer Instanz der Struktur
    struct Complex z;
    // Anlegen eines Zeigers auf eine Struktur
    struct Complex* p = &z;

    // Zugriff auf Elemente der Struktur
    p->re = 1.1;    // z.re
    p->im = 2.2;    // z.im

    return 0;
}
```

Strukturen in der Programmiersprache C
Typ-Neudefinitionen

Verkürzte Schreibeweise bei Strukturen

- Die Instanzerstellung bei Strukturen ist mit sehr viel Schreibarbeit verbunden, da stets die gesamte Deklaration `struct name` verwendet werden muss
- Häufig werden Neudefinitionen mit Hilfe des Schlüsselwortes `typedef` verwendet

Typische Neudefinitionen von Basisdatentypen

```
/* <stdint.h> */
typedef signed char          int8_t;
typedef unsigned char        uint8_t;
typedef short                int16_t;
typedef unsigned short       uint16_t;
typedef int                  int32_t;
typedef unsigned             uint32_t;
typedef long long            int64_t;
typedef unsigned long long   uint64_t;
```

Neudefinition der Struktur mit dem Namen `complex`

```c
#include <stdio.h>

int main()
{
    // Definition einer Struktur mit neuem Namen
    // Syntax: typedef <bekannter Typ> <neuer Typ>
    typedef struct Complex
    {
        double re;
        double im;
    } complex;

    // Anlegen einer Instanz der Struktur
    // durch Verwendung des neuen Typnamens
    complex c;

    // Zugriff auf Elemente der Struktur
    c.re = 1.1;
    c.im = 2.2;

    return 0;
}
```

Verwenden der Struktur

Dynamische Speicherverwaltung in C

Hintergrund zu dynamischer Speicherverwaltung

- In vielen Fällen ist der Bedarf des (Arbeits-)Speichers erst zur Laufzeit eines Programms zu ermitteln, somit ist die benötigte Menge an Speicher zur Entwicklungszeit unbekannt
- Reservierung von Speicher muss sorgfältig erfolgen, da sonst schnell eine Überbelegung erfolgt und ein System nicht mehr funktionsfähig ist

Reservieren und Freigeben von Speicher

- Reservieren von Speicherplatz erfolgt byteweise mit `malloc()`

```
void* malloc (size_t size);
```

- Freigeben von Speicherplatz erfolgt mit free()

```
void free (void* ptr);
```

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

int main()
{
    // Anlegen eines int-Arrays mit 8 Elementen
    int data[8];
    // Ausgabe der Adresse
    printf("%p\n", data);

    // Datenmenge einlesen
    int dataCount = 0;
    scanf("%i", &dataCount);

    // Speicherplatz dynamisch reservieren
    int* dynData;
    dynData = (int*) malloc( sizeof(int) * dataCount );
    // Ausgabe der Adresse
    printf("%p\n", dynData);

    // Speicherplatz freigeben
    free(dynData);

    return 0;
}
```

Anzahl: `dataCount`
Größe Einzelelement: `sizeof(int)`

Ausgabe

```
0xbfdda56b
100
0x797980
```

Warum verschiedene Adressbereiche?

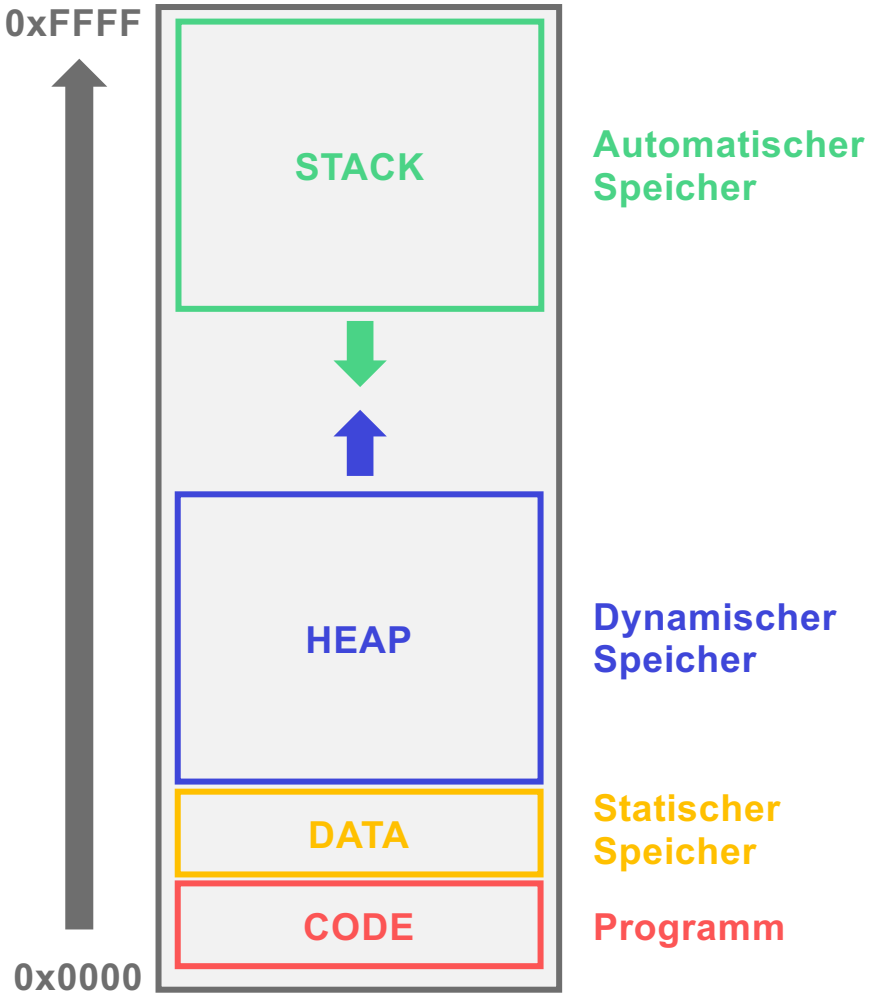
Dynamische Speicherverwaltung in C
Stack vs. Heap

Eine Applikation wird im Hauptspeicher in vier Segmente eingeteilt

- Heap: Bereich in dem Datenelemente dynamisch während der Laufzeit angelegt werden (`malloc, free`)
- Stack: Bereich in dem lokale Variablen, Parameter sowie der Funktionsaufruf abgelegt und automatisch wieder freigegeben werden
- Data: Statischer Speicher, in dem Variablen über die gesamte Laufzeit fest abgelegt sind, wie bspw. `static` Variablen
- Code: Speicher, in dem der eigentliche Code des Programms abgelegt ist

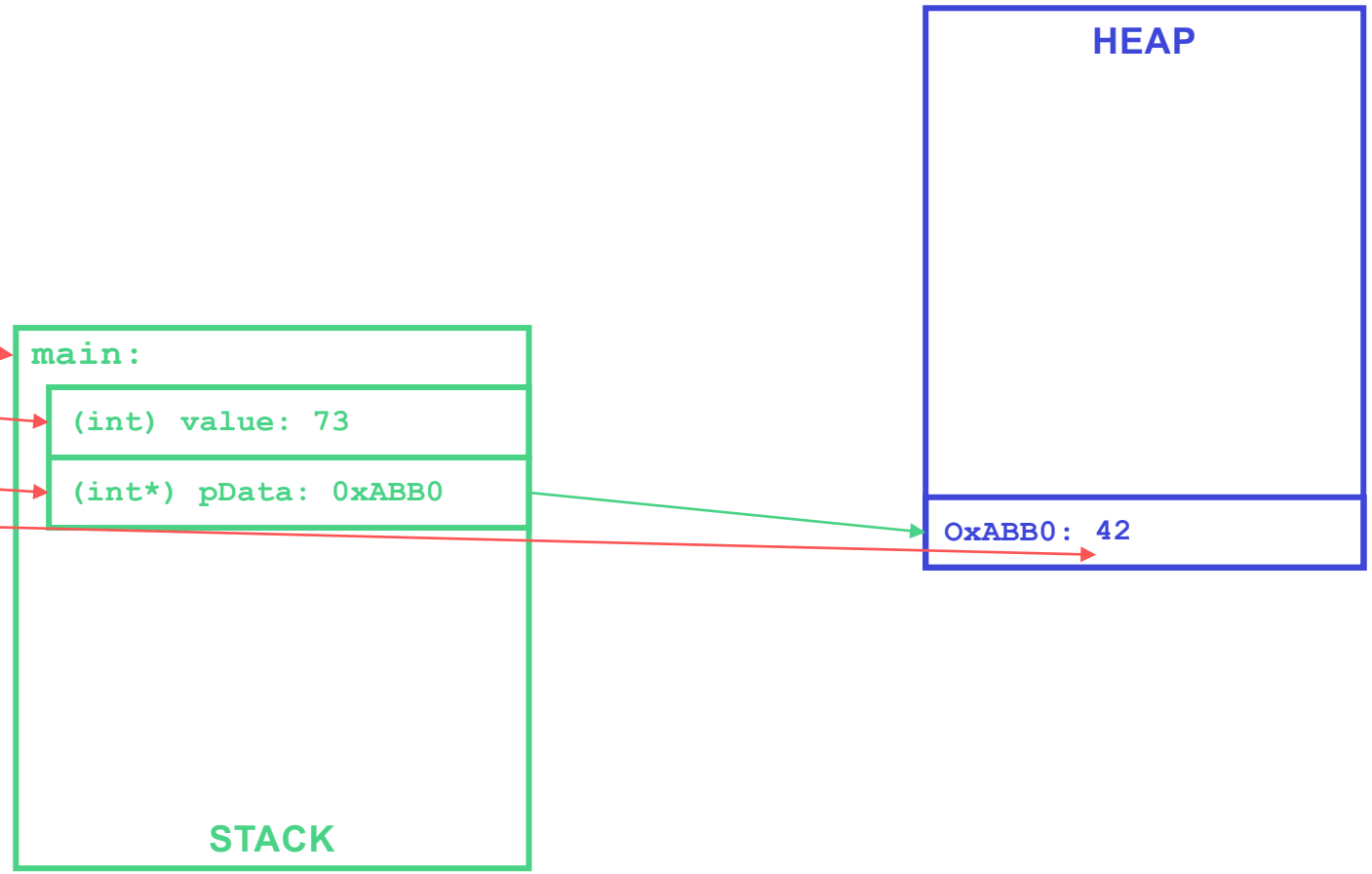
Besonderheiten

- Adressen auf dem Stack nehmen ab
- Adressen auf dem Heap nehmen zu



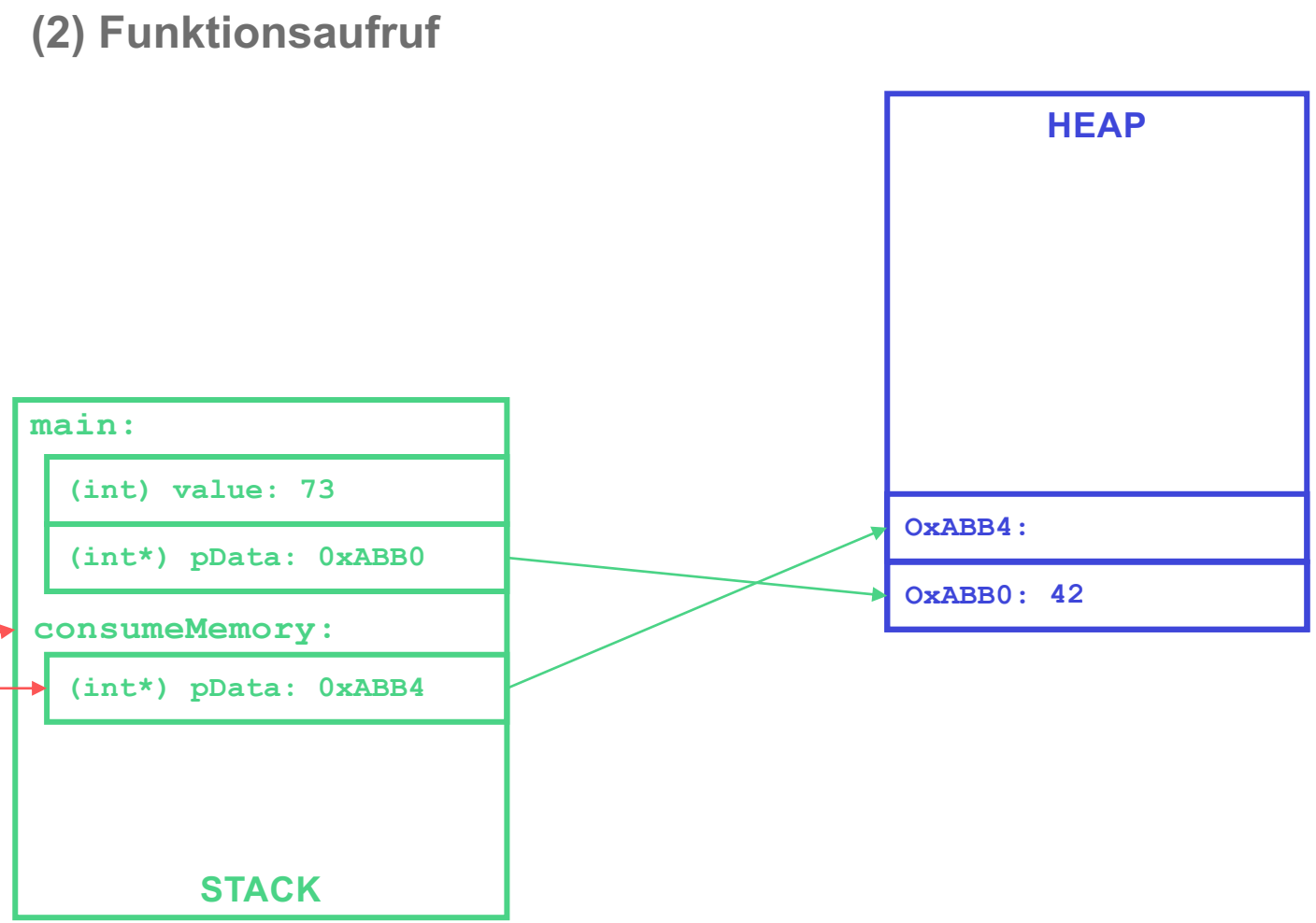
Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

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

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

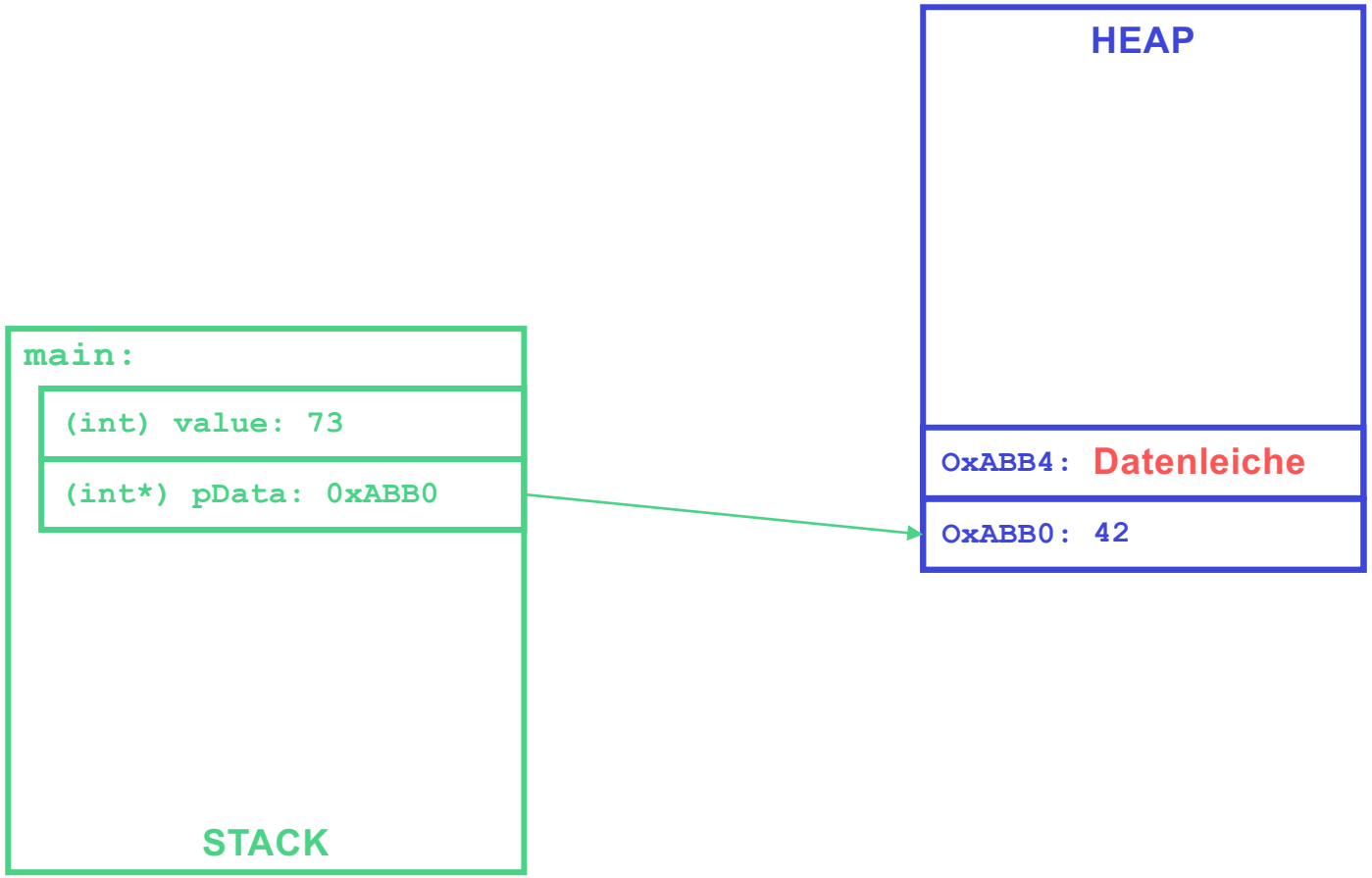
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

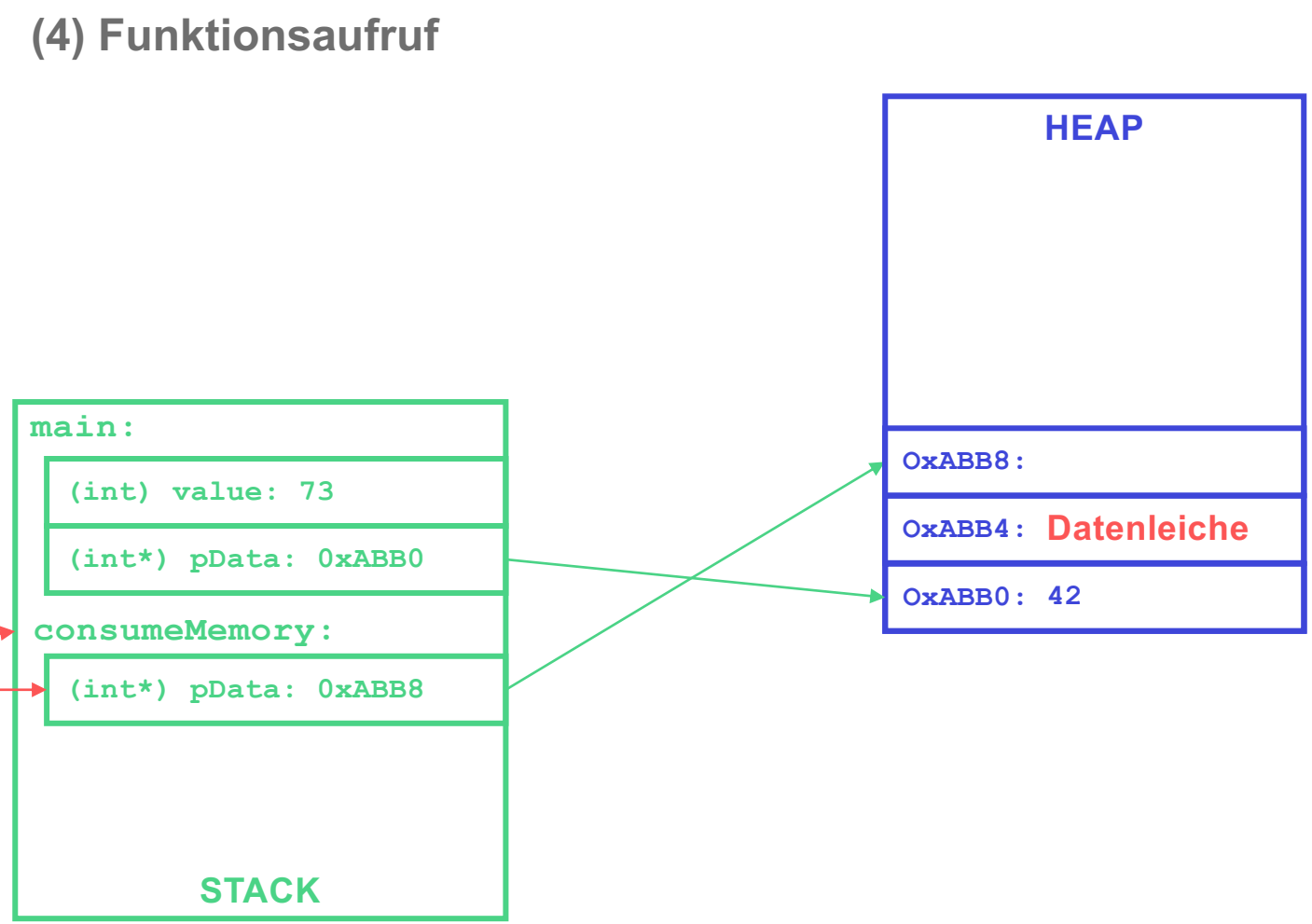
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```

(3) Datenleiche nach Funktionsende



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

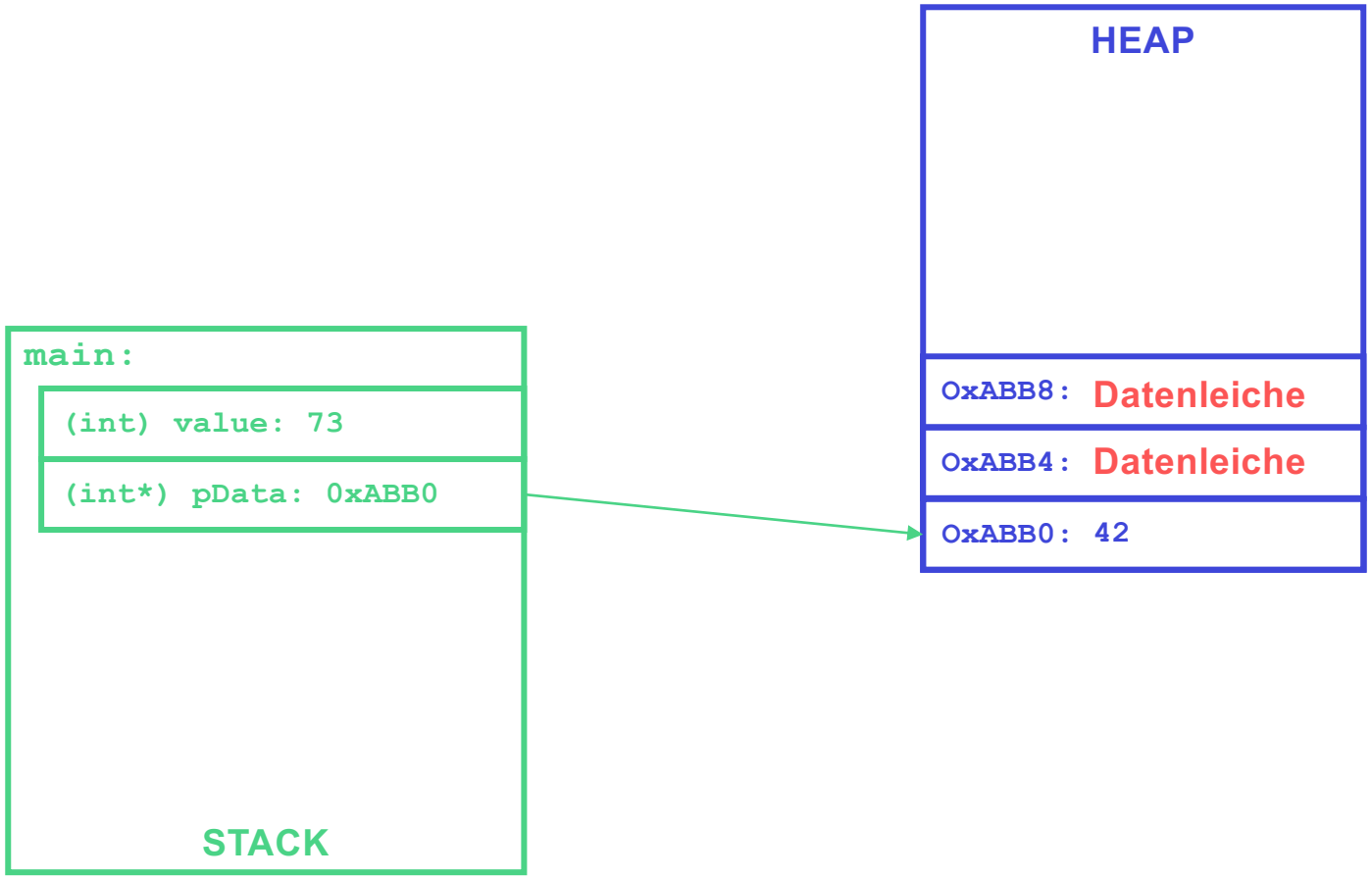
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

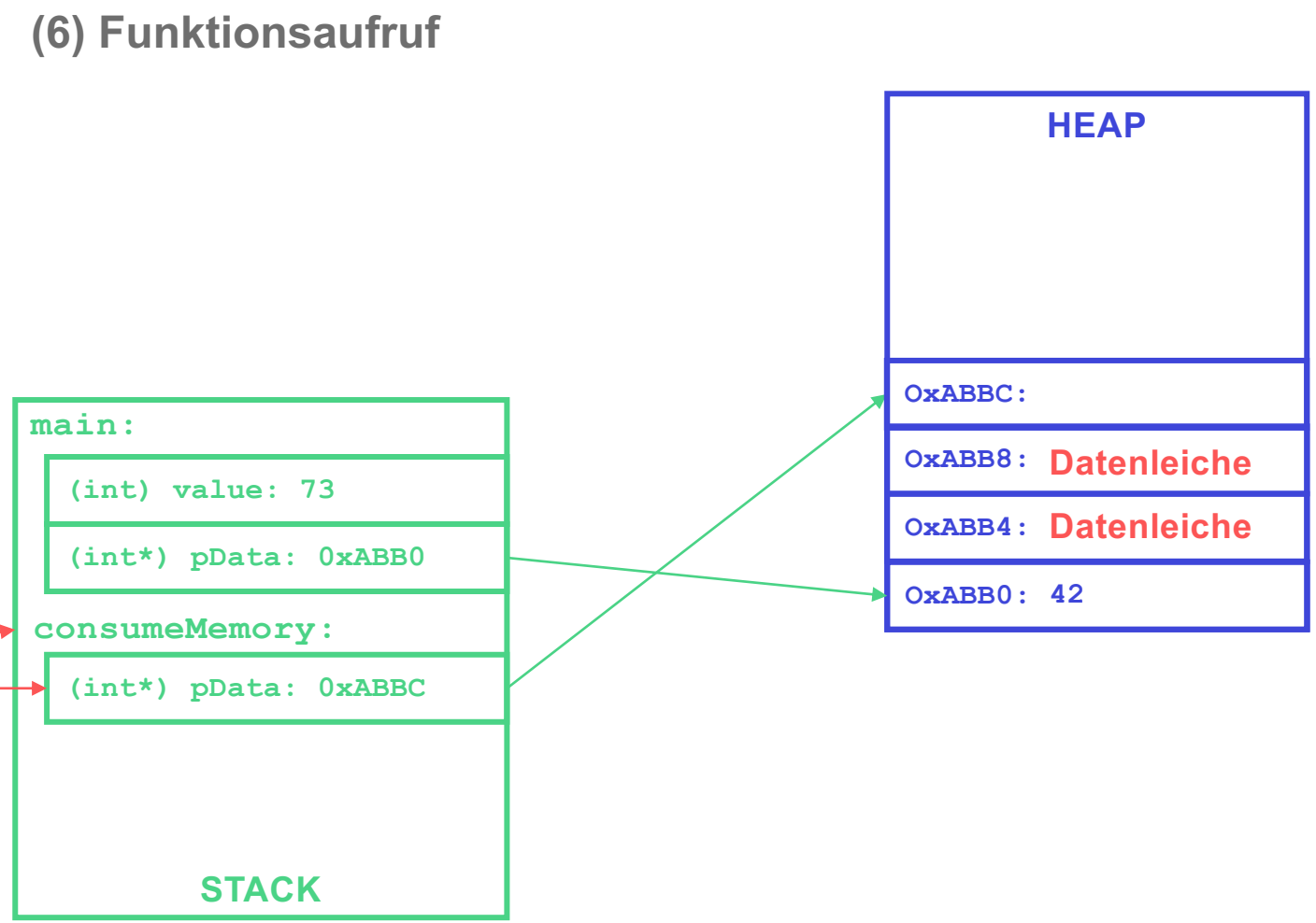
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```

(5) Datenleiche nach Funktionsende



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

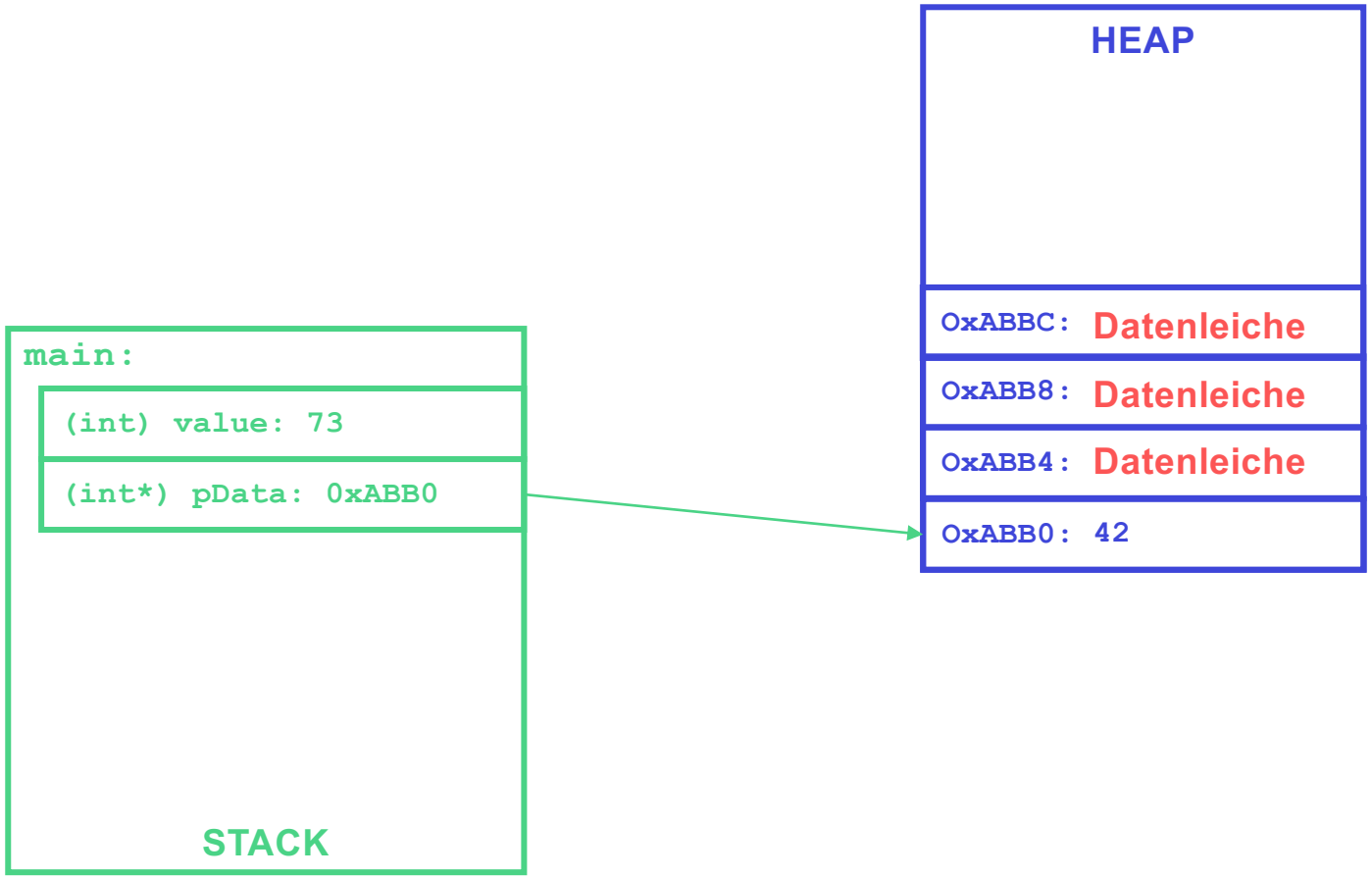
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

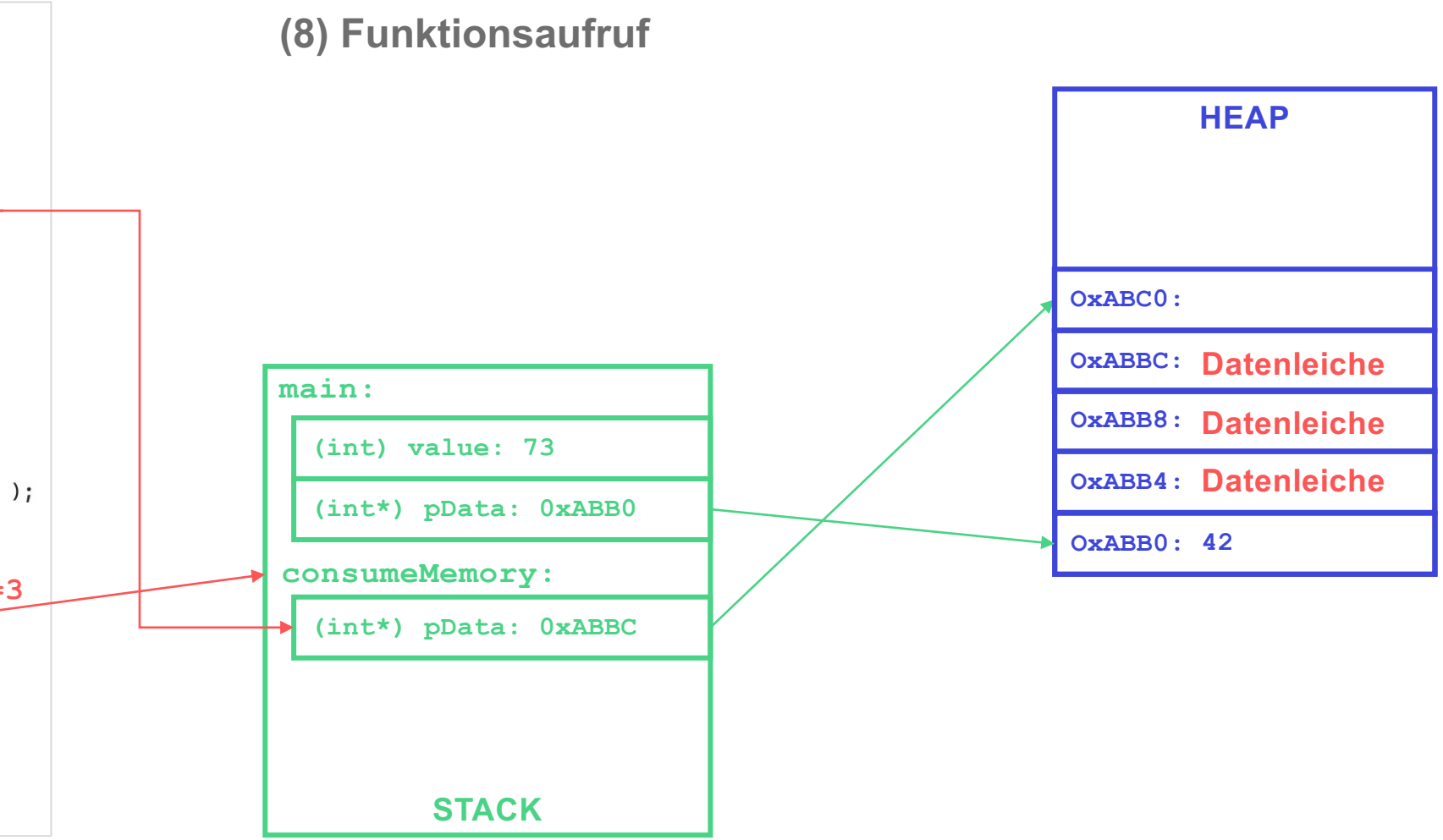
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```

(7) Datenleiche nach Funktionsende



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

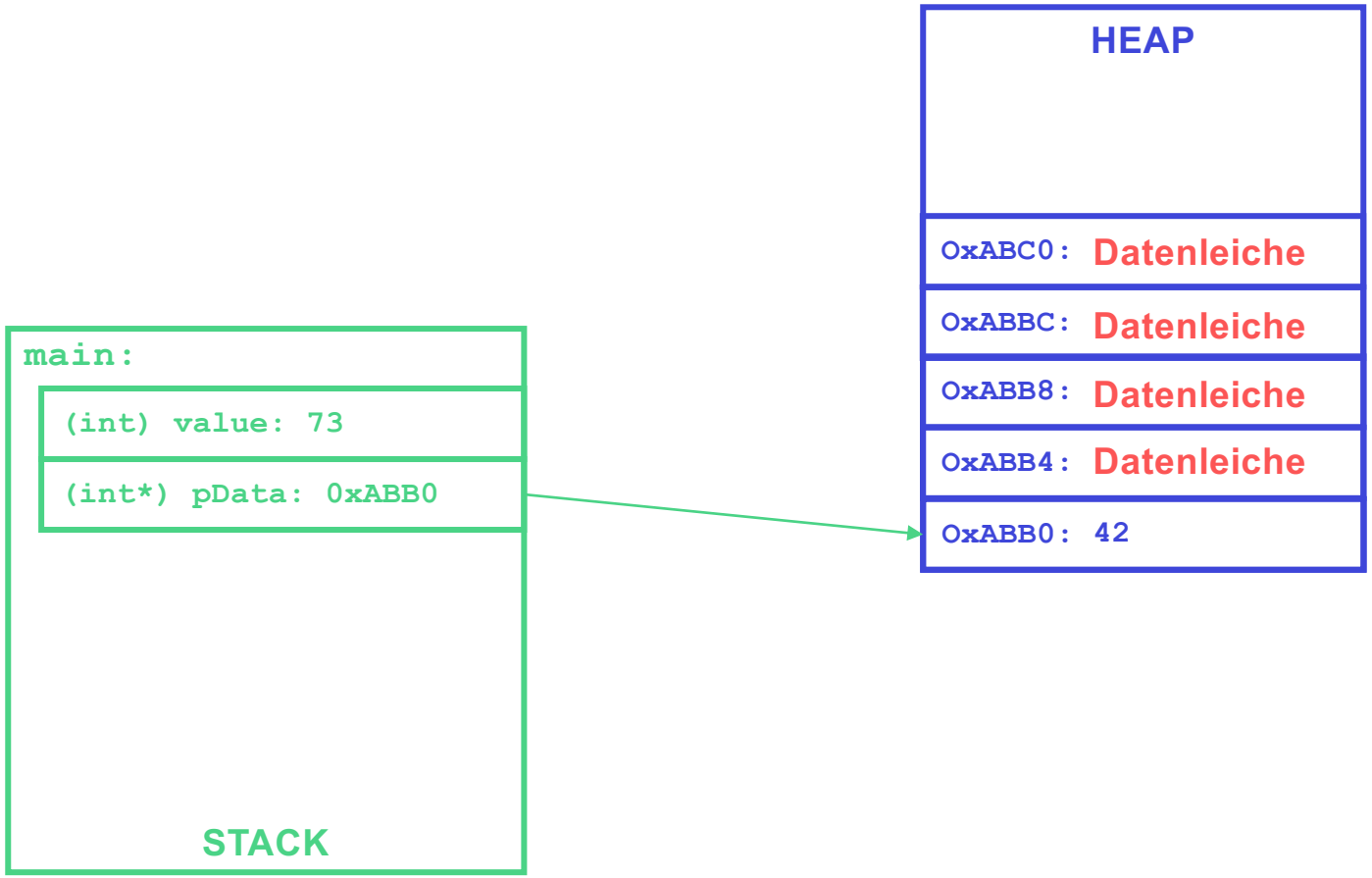
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

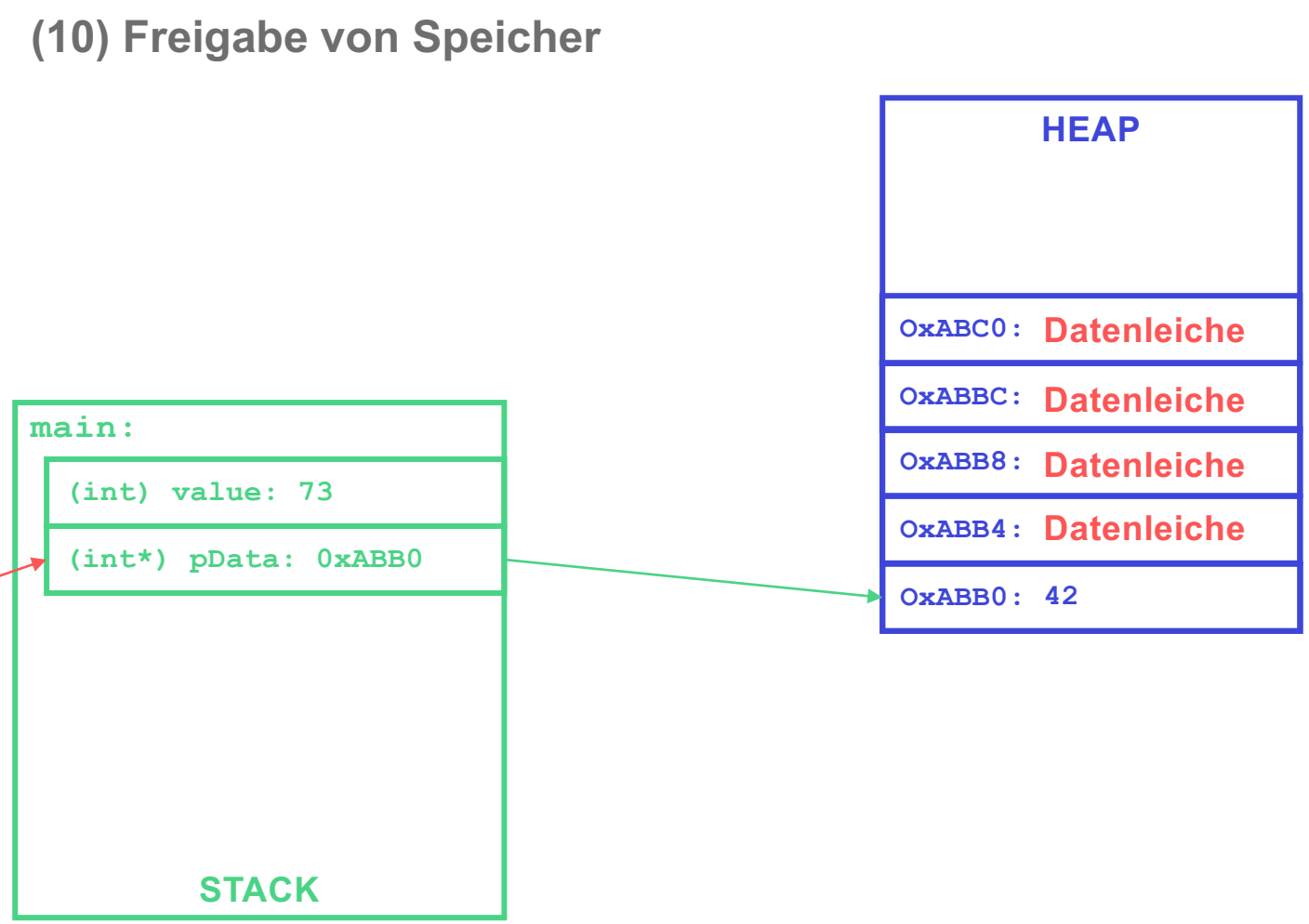
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```

(9) Datenleiche nach Funktionsende



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

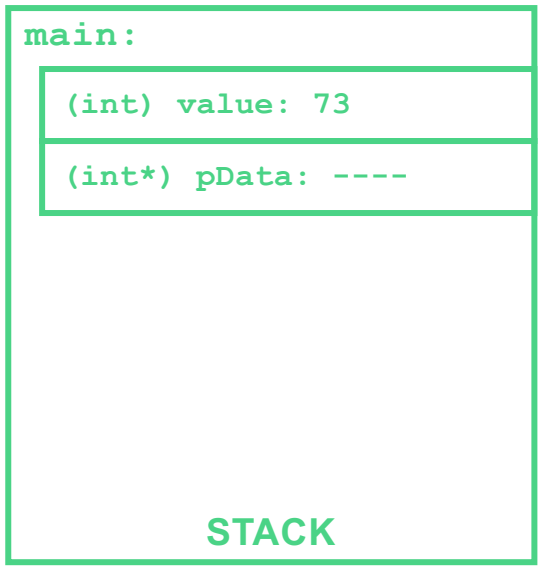
```c
#include <stdio.h>
#include <stdlib.h>

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```

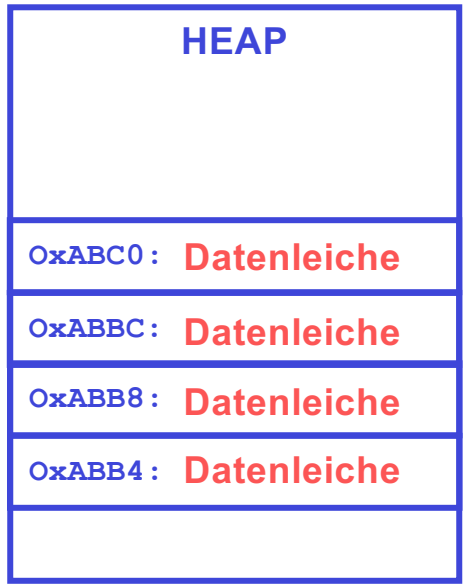


Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

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

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```

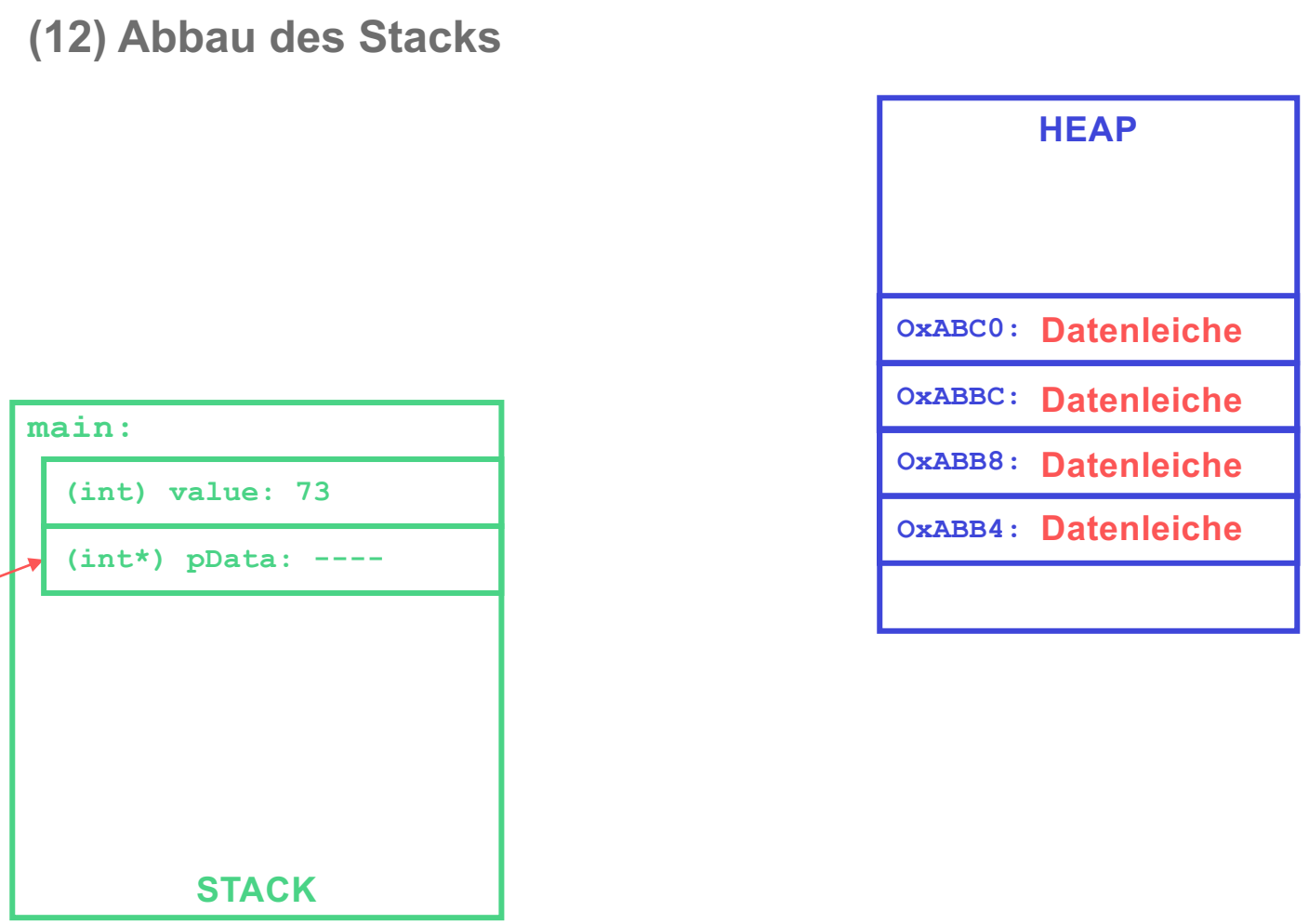
(11) Datum im Heap freigegeben





Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

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

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```



Dynamische Speicherverwaltung in C
Stack vs. Heap an einem Beispiel – Vereinfachte Darstellung

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

// Funktion zum Anlegen von Speicher
void consumeMemory()
{
    int* pData;
    pData = (int*) malloc( sizeof(int) );
}

int main()
{
    // int anlegen
    int value = 73;

    // Speicher reservieren
    int* pData = (int*) malloc( sizeof(int) );
    *pData = 42;

    // Mehrfacher Funktionsaufruf
    for(int i = 0; i < 4; i++)
        consumeMemory();

    // Freigabe von Speicher
    free(pData);

    return 0;
}
```

(13) Datenleichen im Heap
-- hier nur wg. Programmende geleert --

Dynamische Speicherverwaltung in C
Ergänzung zur Freigabe mit free()

Freigabe von mit `malloc()` angelegten Speicherbereich

- Die Funktion `free()` akzeptiert einen Zeiger als Parameter. Dabei kann natürlich nur dann ein Speicher freigegeben werden, wenn dieser auf dem Heap liegt, d.h. wenn dieser mit `malloc()` vorher angelegt wurde
- Wenn `free()` ein NULL Pointer übergeben wird, so hat dies keine Auswirkung

Freigabe von mehreren (zusammenhängenden) Daten

- Die Funktion `malloc()` erlaubt das Reservieren von mehreren Bytes und nicht nur von einzelnen primitiven Datentypen (bspw. `int`)
- Durch den Aufruf von `free()` auf einen Pointer, der auf einen Block im Heap verweist, wird der gesamte Block freigegeben, der mit einem `malloc()` Aufruf reserviert wurde
- In der `malloc.c` Implementierung des GNU Compilers steht dazu (chunk := Block, Datenblock):

```
Minimum overhead per allocated chunk:   4 or 8 bytes
     Each malloced chunk has a hidden word of overhead holding size
     and status information.
```

Dynamische Speicherverwaltung in C

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


// Definition der Struktur
struct Sample
{
    int a;
    double b;
    short c;
};


int main()
{
    // Dynamisches Anlegen einer Struktur
    struct Sample* s = (struct Sample*) malloc(sizeof(struct Sample));
    // Wertzuweisung
    s->a = 5;
    s->b = 4.7;
    s->c = 7;

    // Freigeben der Struktur
    free(s);

    return 0;
}
```

Dynamisches Anlegen einer Struktur

- Erfolgt analog zur dynamischen Speicherverwaltung von primitiven Datentypen mit `malloc()`
- Dynamisch angelegte Daten können nur über einen Zeiger angesprochen werden
- Alternativer Zugriff über Dereferenzierung

  `s->a = 5;` bzw. `(*s).a = 5;`

Dynamische Speicherverwaltung in C
Byteweises Kopieren von Daten

Erstellen von 1:1 Kopien (binär)

- Häufig müssen existierende Daten kopiert werden, um den ursprünglichen Datensatz nicht zu verändern
- Dieser Mechanismus ist für alle Datentypen möglich, lohnt sich jedoch insbesondere für große Strukturen (arrays, structs)

Kopieren von Speicherbereichen in C

- Anwenden der Funktion `memcpy()`

```
void* memcpy(void* dest, const void* src, size_t n);
```

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


// Definition der Struktur
typedef struct Sample
{
    int a;
    double b;
    short c;
} sample;


int main()
{
    // Anlegen einer Struktur
    sample s;
    s.a = 5;
    s.b = 7.0;
    s.c = 3;

    // Dynamisches Anlegen einer Struktur
    sample* ps = (sample*) malloc(sizeof(sample));

    // Byteweises Kopieren
    memcpy(ps, &s, sizeof(sample));
    printf("%i %lf %i\n", ps->a, ps->b, ps->c);

    // Freigeben der Struktur
    free(ps);

    return 0;
}
```

Zusammenfassung

Verwenden von Strukturen

- Definition mit `struct`, Typneudefinition mit `typedef`
- Verwendung auch durch Zeiger (-> Operator)
- Kopieren von Strukturen ebenfalls mit `memcpy()`
- Speicherbelegung von Strukturen hängt von Reihenfolge der Elemente ab

Dynamische Speicherverwaltung

- Programme sind in vier Segmente aufgeteilt: Stack, Heap, Data, Code
- Dynamisch reservierter Speicher muss wieder freigegeben werden (sonst Datenleiche)