Pointer, Arrays und New

Florian Adamsky, B. Sc. (PhD cand.)

florian.adamsky@iem.thm.de
http://florian.adamsky.it/

Softwareentwicklung im WS 2014/15

THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Outline

Wiederholung der letzten Vorlesung

- C++ ist maschinennah, schnell und unterstützt das objektorientierte Paradigma
- eine Klasse ist ein Bauplan und kann in C++ über das Schlüsselwort `class` erzeugt werden
- eine Klasse besitzt Attribute und Methoden
- von einer Klassen können viele Objekte erzeugt werden
- der Zugriff auf die Attribute und Methoden eines Objekt erfolgt über den Punktoperator `.`
- Objekte können über Methoden Nachrichten austauschen
- Datenkapselung ist kann über die Zugriffsschlüsselwörter `private` und `public` in C++

Inhaltsverzeichnis

Was ist ein Pointer/Zeiger?

Definition (Pointer)

Für einen Typ `T`, ist `T*` ein Zeiger auf `T`. Ein Pointer (Zeiger) ist eine Variable die eine Speicheradresse speichert und dadurch auf eine andere Variable oder Objekt zeigt. (Analogie: Verknüpfung in Windows)

```
int  x   = 5;
int* ptr = &x;
```

Was ist ein Pointer/Zeiger?

Definition (Pointer)

Für einen Typ `T`, ist `T*` ein Zeiger auf `T`. Ein Pointer (Zeiger) ist eine Variable die eine Speicheradresse speichert und dadurch auf eine andere Variable oder Objekt zeigt. (Analogie: Verknüpfung in Windows)

```
int  x   = 5;
int* ptr = &x;
```

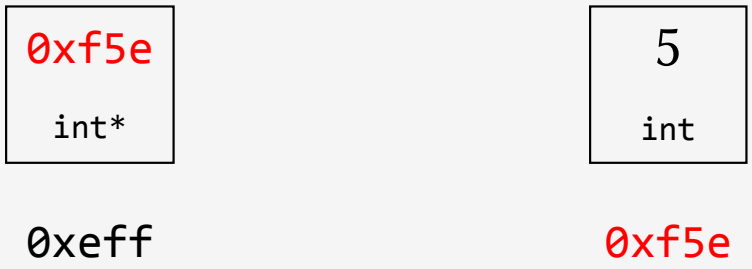
0xf5e

Was ist ein Pointer/Zeiger?

Definition (Pointer)

Für einen Typ `T`, ist `T*` ein Zeiger auf `T`. Ein Pointer (Zeiger) ist eine Variable die eine Speicheradresse speichert und dadurch auf eine andere Variable oder Objekt zeigt. (Analogie: Verknüpfung in Windows)

```
int  x   = 5;
int* ptr = &x;
```



Was ist ein Pointer/Zeiger?

Definition (Pointer)

Für einen Typ `T`, ist `T*` ein Zeiger auf `T`. Ein Pointer (Zeiger) ist eine Variable die eine Speicheradresse speichert und dadurch auf eine andere Variable oder Objekt zeigt. (Analogie: Verknüpfung in Windows)

```
int  x   = 5;
int* ptr = &x;
```

Dereferenzierung

Definition (Dereferenzierungs-Operator)

Der *-Operator gibt den Wert aus, auf dem der Pointer zeigt.

Example (Beispiel)

```
int  x   = 5;
int* ptr = &x;

// Integer
cout <<  x << endl;      // 5
cout << &x << endl;      // 0xf5e

// Pointer
cout << *ptr << endl;    // 5
cout <<  ptr << endl;    // 0xf5e
```

Was ist eine Referenz?

Definition (Referenz)

Referenz ist ein Pointer mit automatischer Dereferenzierung. Der Unterschied ist, das Referenzen nach der Initialisierung nicht mehr verändert werden können.

```
int  x   = 5;
int& ptr = x;
```

Was ist eine Referenz?

Definition (Referenz)

Referenz ist ein Pointer mit automatischer Dereferenzierung. Der Unterschied ist, das Referenzen nach der Initialisierung nicht mehr verändert werden können.

```
int  x   = 5;
int& ptr = x;
```

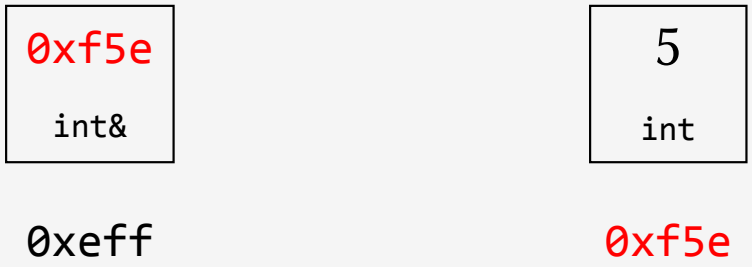
0xf5e

Was ist eine Referenz?

Definition (Referenz)

Referenz ist ein Pointer mit automatischer Dereferenzierung. Der Unterschied ist, das Referenzen nach der Initialisierung nicht mehr verändert werden können.

```
int  x   = 5;
int& ptr = x;
```



Was ist eine Referenz?

Definition (Referenz)

Referenz ist ein Pointer mit automatischer Dereferenzierung. Der Unterschied ist, das Referenzen nach der Initialisierung nicht mehr verändert werden können.

```
int  x   = 5;
int& ptr = x;
```

Pointer/Referenzen

	Variable	Pointer	Referenzen
Initialisierung	`int x = 5;`	`int* ptr = &x;`	`int& ref = x`
Wert	`x`	`*ptr`	`ref`
Speicheradresse	`&x`	`ptr`	`&ref`

Vergleich Pointer und Referenzen

Pointer/Zeiger

- Arithmetische Operationen wie `ptr++` sind erlaubt
- Pointer müssen nicht initialisiert werden
- Können ihren Wert verändert, dadurch auf verschiedene Variablen zeigen

Referenz

- Arithmetische Operationen sind sind nicht erlaubt
- Referenzen müssen initialisiert werden
- Können nicht verändert werden, dadurch zeigen sie immer auf die selbe Variable

Call-By-Value

Definition (Call-By-Value)

Die Parameterübergabe *Call-By-Value* kopiert Argumente eines Funktions- oder Methodenaufrufs in neue Variablen.

Example (Call-By-Value)

```
void tauschen(int x, int y) {
    int temp = x;       // speicher den Wert von x zwischen
    x = y;              // überschreibe x mit dem Wert von y
    y = temp;           // überschreibe y mit dem Wert von temp
}
// ...
int a = 23, b = 42;
tauschen(a, b);
cout << a << " " << b << endl;
```

Call-By-Value

Definition (Call-By-Value)

Die Parameterübergabe *Call-By-Value* kopiert Argumente eines Funktions- oder Methodenaufrufs in neue Variablen.

Example (Call-By-Value)

```
void tauschen(int x, int y) {
    int temp = x;       // speicher den Wert von x zwischen
    x = y;              // überschreibe x mit dem Wert von y
    y = temp;           // überschreibe y mit dem Wert von temp
}
// ...
int a = 23, b = 42;
tauschen(a, b);
cout << a << " " << b << endl;
```

a = 23 und b = 42 keine Veränderung

Call-By-Reference

Definition (Call-By-Reference)

Die Parameterübergabe *Call-By-Reference* kopiert die Speicheradresse der Argumente in neue Referenzen.

Example (Call-By-Reference)

```
void tauschen(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
// ...
int a = 23, b = 42;
tauschen(a, b);
cout << a << " " << b << endl;
```

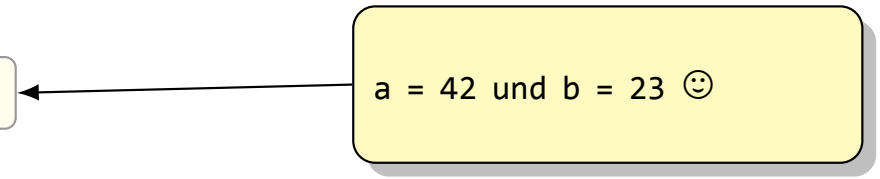
Call-By-Reference

Definition (Call-By-Reference)

Die Parameterübergabe *Call-By-Reference* kopiert die Speicheradresse der Argumente in neue Referenzen.

Example (Call-By-Reference)

```
void tauschen(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
// ...
int a = 23, b = 42;
tauschen(a, b);
cout << a << " " << b << endl;
```



Pointer-Based Call-By-Reference

Definition (Call-By-Pointer)

Die Parameterübergabe *Pointer-Based Call-By-Reference* kopiert die Speicheradresse der Argumente in Pointer/Zeiger.

Example (Pointer-Based Call-By-Reference)

```
void tauschen(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = *temp;
}
// ...
int a = 23, b = 42;
tauschen(&a, &b);
cout << a << " " << b << endl;
```

Pointer-Based Call-By-Reference

Definition (Call-By-Pointer)

Die Parameterübergabe *Pointer-Based Call-By-Reference* kopiert die Speicheradresse der Argumente in Pointer/Zeiger.

Example (Pointer-Based Call-By-Reference)

```
void tauschen(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = *temp;
}
// ...
int a = 23, b = 42;
tauschen(&a, &b);
cout << a << " " << b << endl;
```

a = 42 und b = 23 ☺

Inhaltsverzeichnis

Was ist ein Array?

- strukturierter Datentyp
  - enthält ein oder mehrere Elemente des gleichen Datentyps
  - die Elemente sind sequenziell im Speicher angeordnet
- Definition eines Arrays ist wie folgt:

```
int array[10];
```

- Anzahl der Elemente muss zur Compilezeit feststehen
- Zugriff auf die einzelnen Elemente erfolgt über `[]` und fängt bei 0 an

```
std::cout << array[0] << std::endl;
```

- Initialisierung erfolgt über:

```
int array[]  = {'T', 'H', 'M'};
int array[3] = {'T', 'H', 'M'};
```

Array im Speicher

Example (Array-Initialisierung)

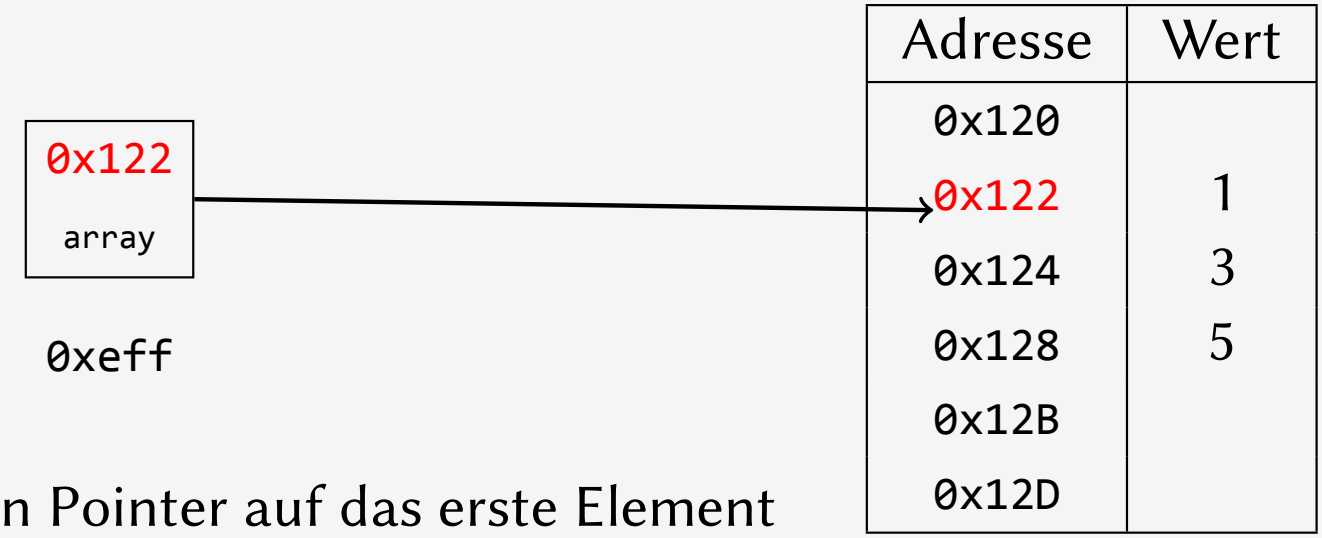
```
int array[] = {1, 3, 5};
```

Adresse	Wert
0x120	
0x122	1
0x124	3
0x128	5
0x12B	
0x12D	

Array im Speicher

Example (Array-Initialisierung)

```
int array[] = {1, 3, 5};
```



Name des Arrays ist ein Pointer auf das erste Element

Arrays als Funktions- oder Methodenparameter

Example (Funktionsparameter)

```
#include <iostream>

void array_ausgabe(int a[], int size) {
   for (int i = 0; i < size; i++) {
       std::cout << a[i] << std::endl;
   }
}

int main() {
    int a[5] = {10, 20, 30, 40, 50};

    array_ausgabe(a, 5);
}
```

Arrays als Funktions- oder Methodenparameter

Example (Funktionsparameter)

```
#include <iostream>

void array_ausgabe(int a[], int size) {
   for (int i = 0; i < size; i++) {
       std::cout << a[i] << std::endl;
   }
}

int main() {
    int a[5] = {10, 20, 30, 40, 50};

    array_ausgabe(a, 5);
}
```

Übergabe eines Pointers auf das erste Element und die Größe des Arrays

Arrays als Funktions- oder Methodenparameter

Example (Funktionsparameter)

```
#include <iostream>

void array_ausgabe(int a[], int size) {
    for (int i = 0; i < size; i++) {
        std::cout << a[i] << std::endl;
    }
}

int main() {
    int a[5] = {10, 20, 30, 40, 50};

    array_ausgabe(a, 5);
}
```

Der Compiler übersetzt `int a[]` zu `int a*`

Arrays und Pointer-Arithmetik

- Name des Arrays ist ein Pointer auf das erste Element
- Pointer kann man rechnen, dabei kommt es auf den Datentyp drauf an

Example (Addition)

```
if (array[2] == *(array + 2)) {
    cout << "Der Ausdruck is gleich" << endl;
}
```

Example (Subtraktion)

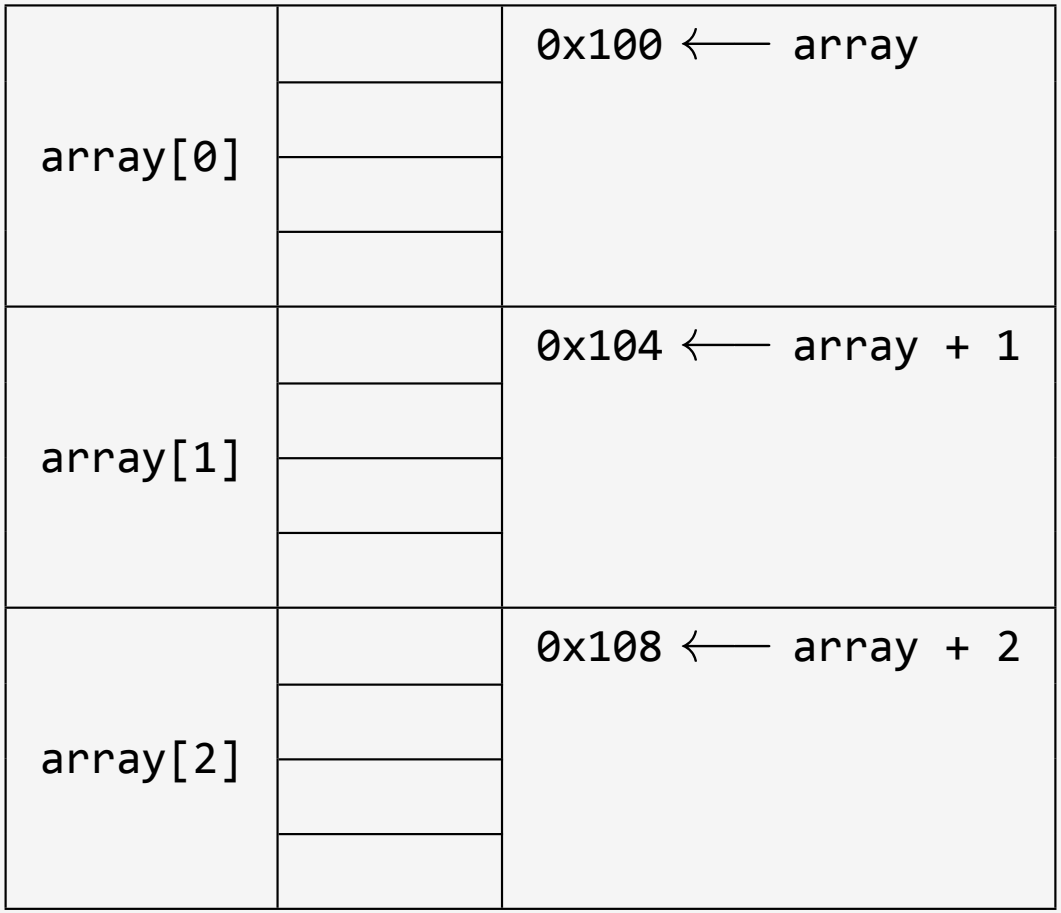
```
int array[10] ;
int* ptr1 = array + 2 ;
int* ptr2 = array + 5 ;

cout << (ptr2 - ptr1) ; // Gibt 3 aus
cout << (ptr1 - ptr3) ; // Gibt −3 aus
```

Arrays und Pointer-Arithmetik (Int)

Example (Array-Initialisierung)

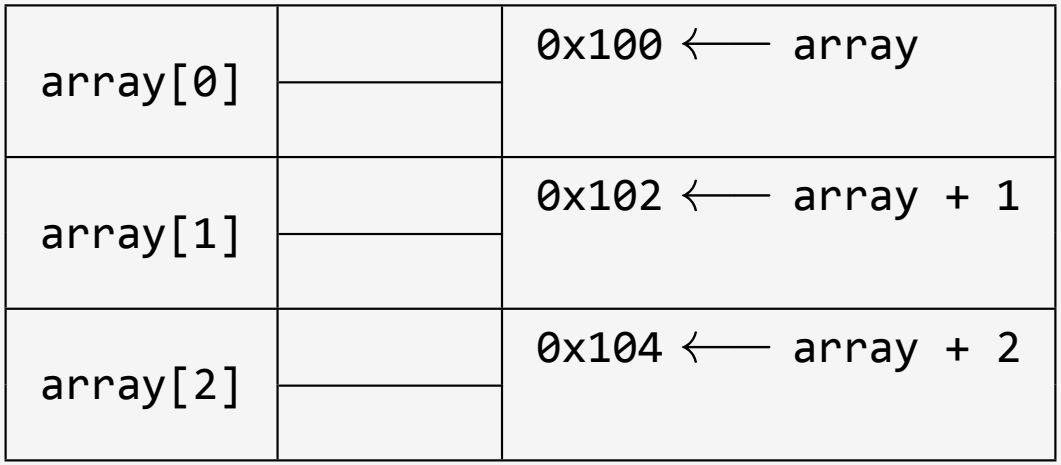
```
int array[3]; // Int ist in der Regel 4 Bytes groß
```



Arrays und Pointer-Arithmetik (Short)

Example (Array-Initialisierung)

```
short array[3]; // Short ist in der Regel 2 Bytes groß
```



Mehrdimensionale Arrays

- Arrays können n Dimensionen haben
  - 2-dimensionales Array wird wie folgt definiert

```
// 4 Zeilen 5 Spalten
int matrix[4][5] = { {10,20,30,40,50},
                     {15,25,35,45,55},
                     {20,30,40,50,60},
                     {25,35,45,55,65}};
```

  - Zugriff auf einzelne Elemente über `[Reihe][Spalte]`

```
cout << matrix[2][3] << endl;   // gibt 50 aus
```

Inhaltsverzeichnis

Einleitung

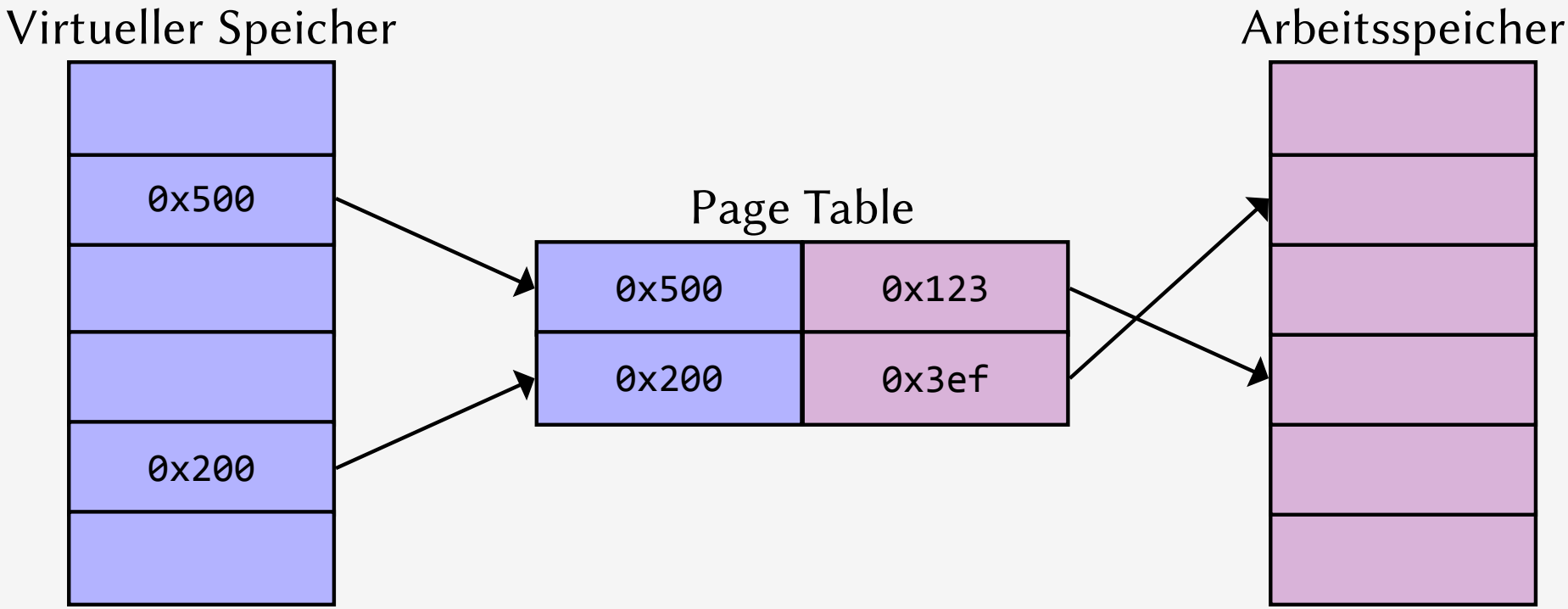
- Arbeitsspeicher, Hauptspeicher, Random Access Memory (RAM)
- Enthält temporäre Daten
- Flüchtig → Daten gehen nach dem Ausschalten verloren
- Hohe Geschwindigkeit im Vergleich zur Festplatte
  - Datenraten: 6–30 GByte/s

Abbildung: Bild von Eric Hamiter

Speicherverwaltung

- Kein Programm hat direkten Zugriff auf den Arbeitsspeichern, sondern nur auf *virtuellen Speicher*
- Die *virtuelle Speicherverwaltung* wird von nahezu allen Betriebssystemen verwendet

Virtuelle Speicherverwaltung 1



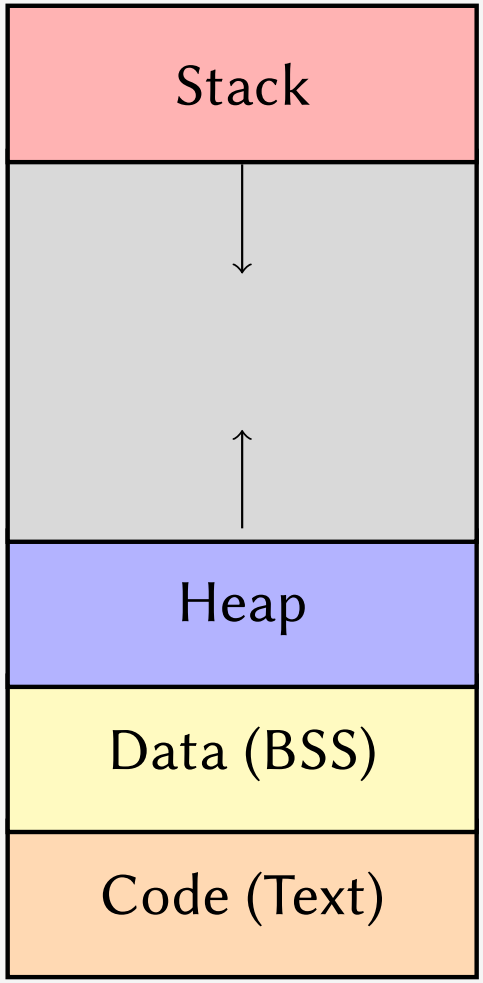
Virtuelle Speicherverwaltung 2

Vorteile

- Das Betriebssystem kümmert sich um das Speichermanagement
- Jeder Prozess darf nur seinen Speicher sehen
- Arbeitsspeicher kann über die Festplatte erweitert werden (SWAP)

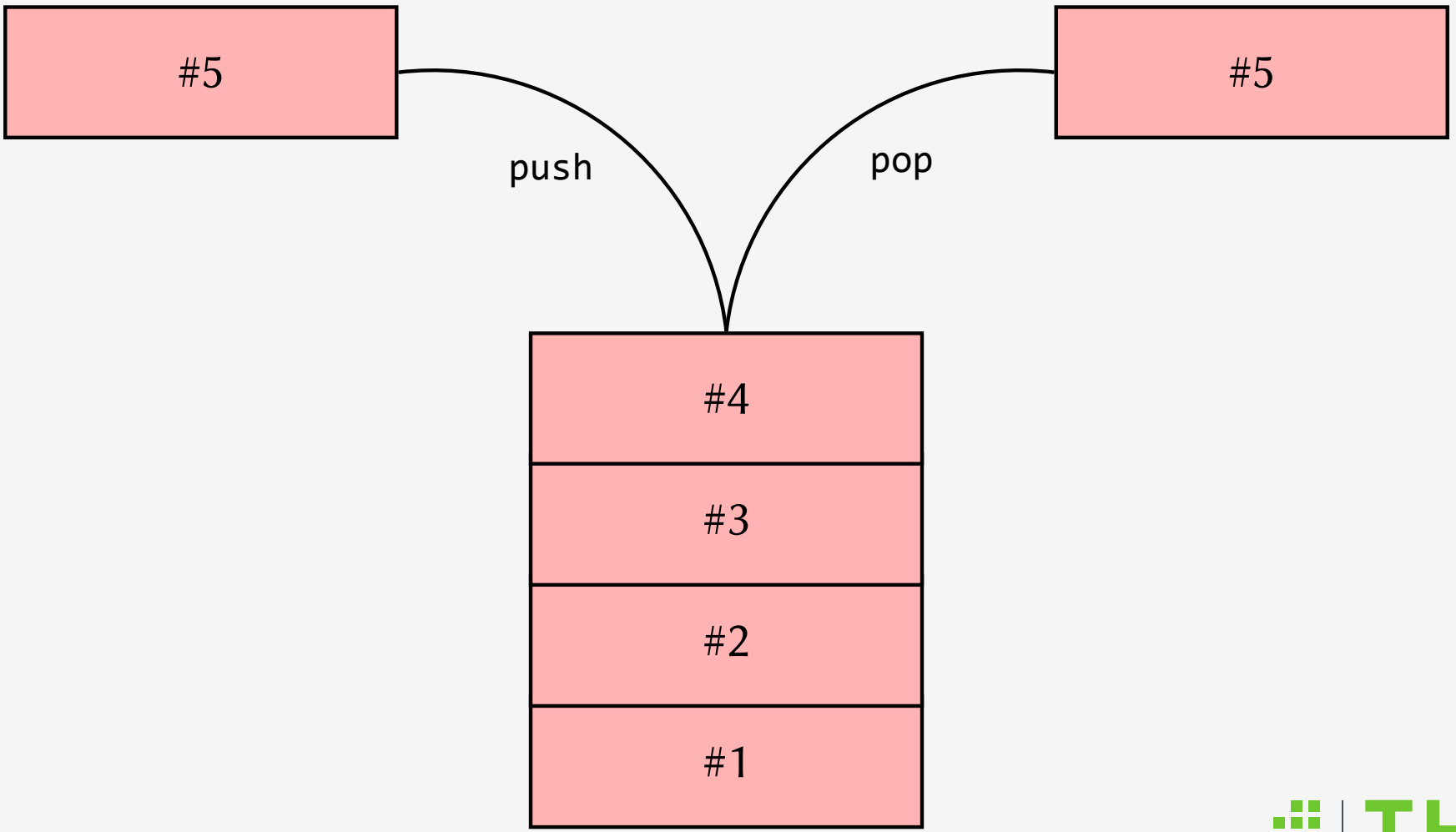
Stack und Heap

- Stack (strukturiert)
  - Funktions/Methoden Parameter
  - Lokale Variablen
  - Funktions/Methoden Rückgabewerte
- Heap (unstrukturiert)
  - dynamischer Speicher
- Data
  - Statische und globale Variablen
- Code (Text)
  - enthält die ausführbaren Instruktionen
  - schreibgeschützt



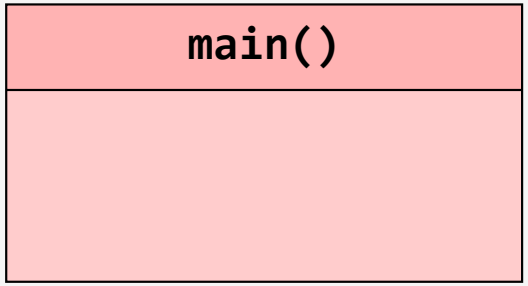
Stack (Datenstruktur)

- Stack ist eine Last-in-First-out (LIFO) Datenstruktur



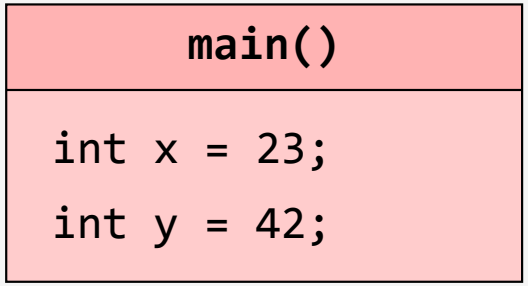
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



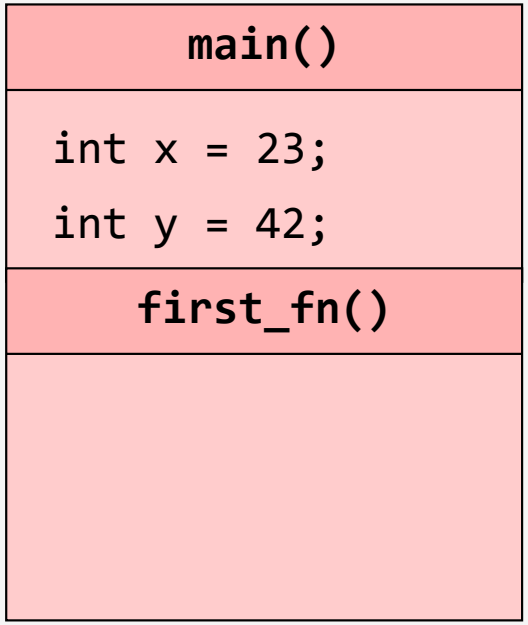
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



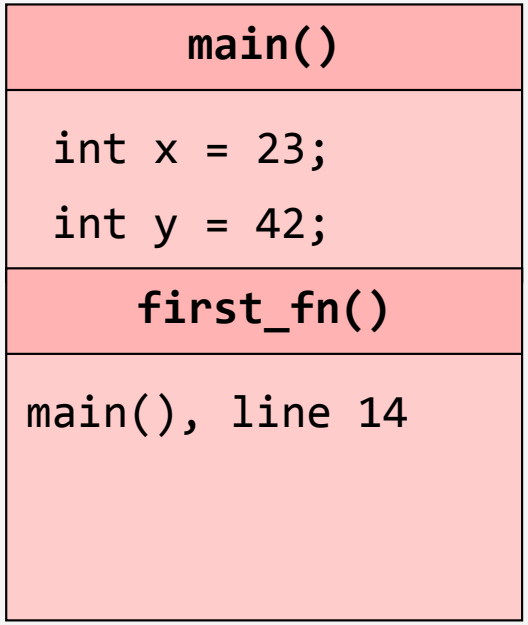
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



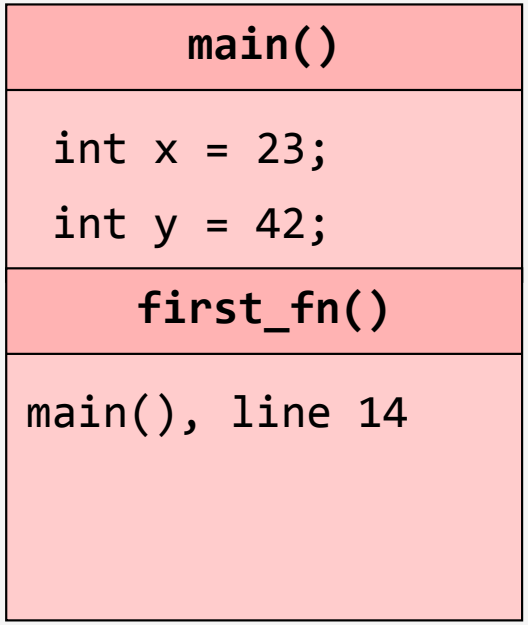
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



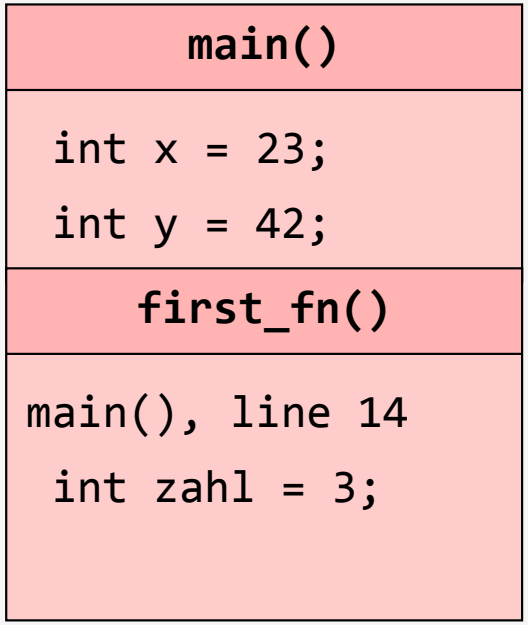
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



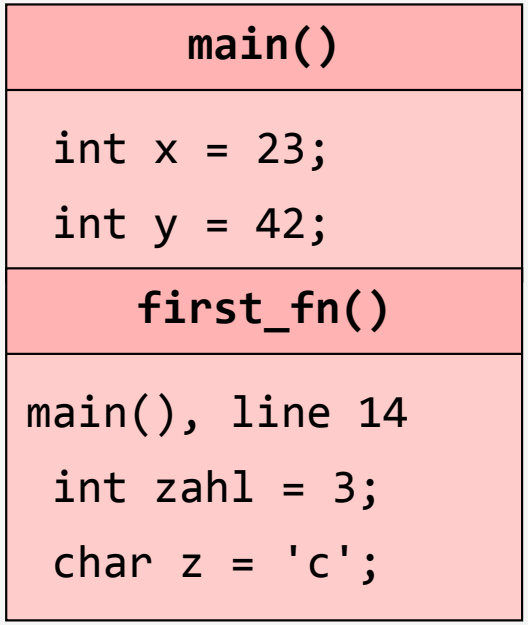
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



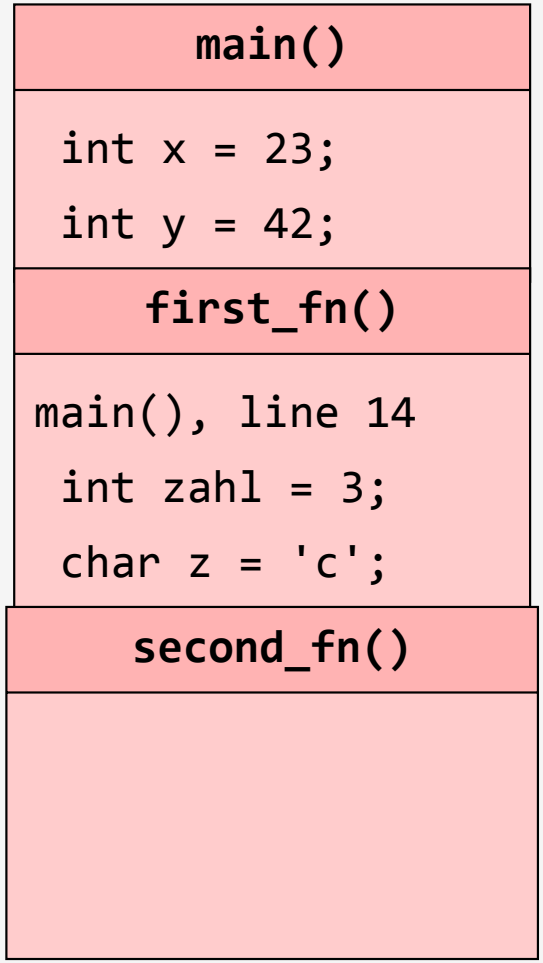
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



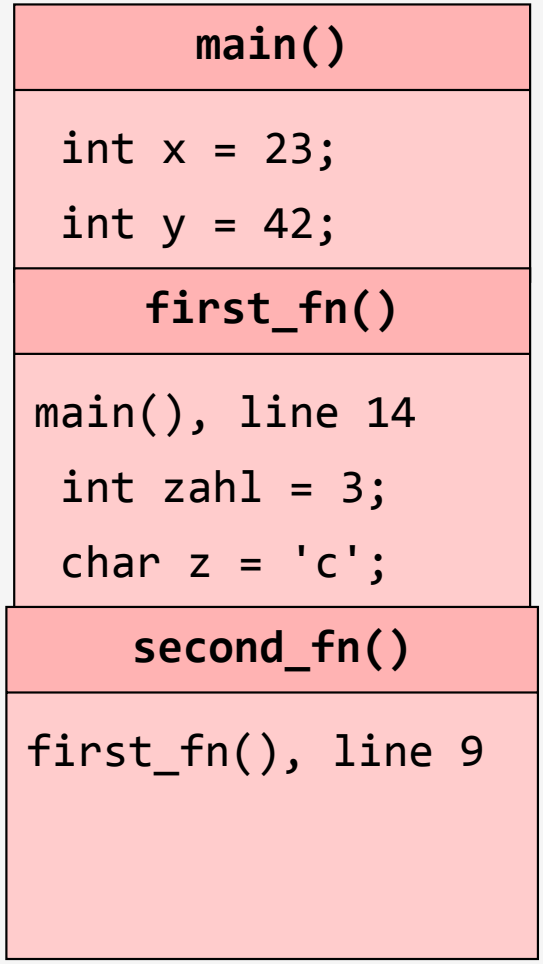
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



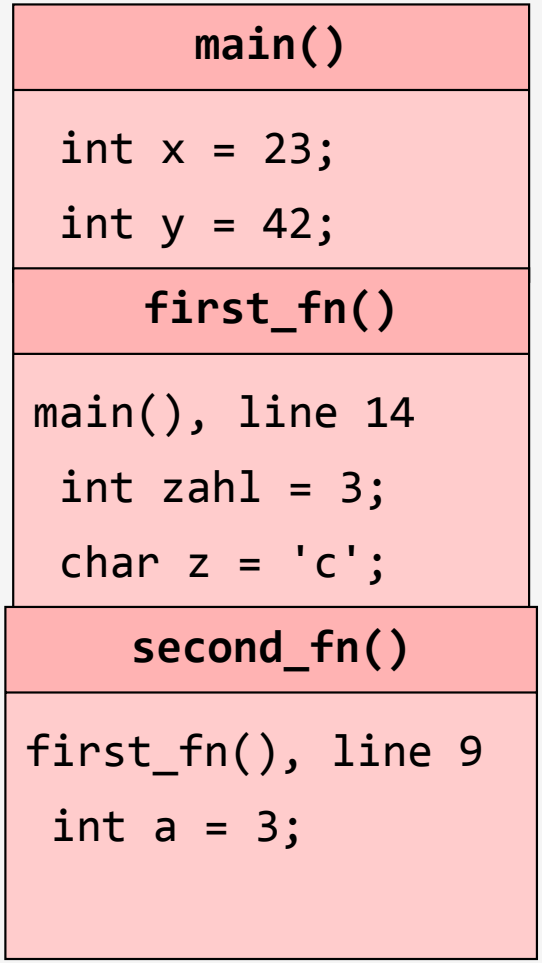
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



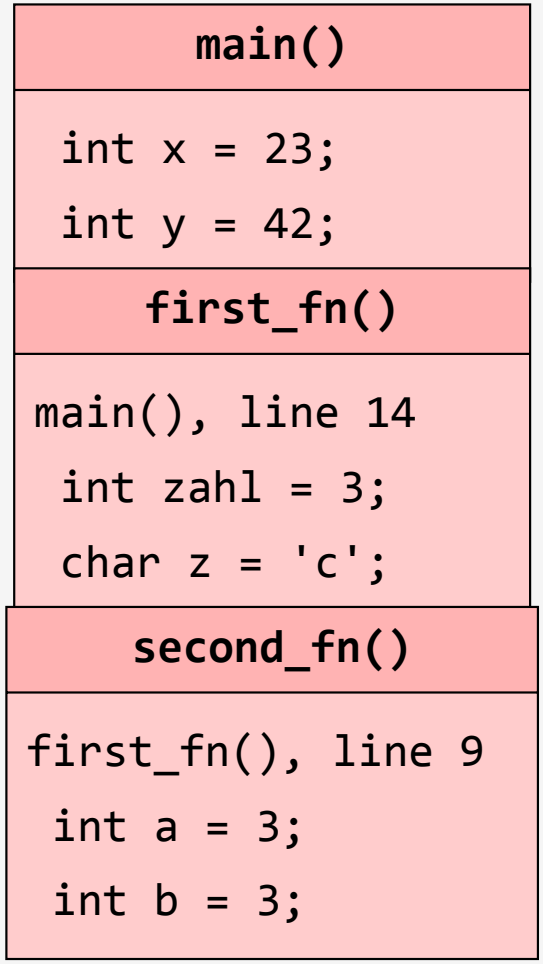
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



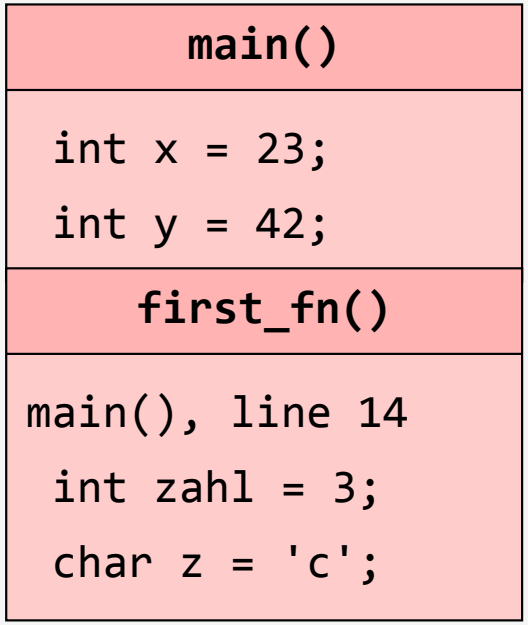
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



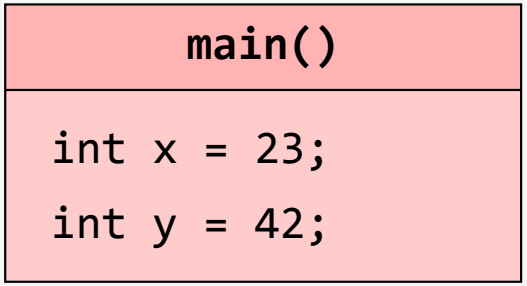
Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



Stack

```
1  void second_fn(int a) {
2      int b = a;
3  }
4
5  void first_fn(void) {
6      int  zahl = 3;
7      char z    = 'c';
8      second_fn(zahl);
9  }
10
11 int main(void) {
12     int x = 23, y = 42;
13     first_fn();
14 }
```



Stack Overflow

- Stack hat nur eine begrenzte Größe
- Falls der Speicher aufgebraucht ist, führt das zu einem *Stack Overflow*
  - weitere Alloziierungen *schwappen* in andere Speicherbereiche über

Stack Overflow

- Stack hat nur eine begrenzte Größe
- Falls der Speicher aufgebraucht ist, führt das zu einem *Stack Overflow*
  - weitere Alloziierungen *schwappen* in andere Speicherbereiche über

Stack Overflow

```
#include <iostream>

int main(void) {
    int stack[100000000]; // Int-Array mit 100.000.000 Eintraegen
}
```

Vor- und Nachteile des Stacks

Vorteile

- ProgrammierInn braucht sich nicht um die Aufräumarbeiten kümmern
- Keine Pointer notwendig, da der Zugriff direkt erfolgt
- dadurch schneller

Nachteile

- Größe muss zur Compilezeit feststehen
- Stack ist nicht besonders groß $\rightarrow$ Stack Overflow
- langsamer bei *call-by-value* da viel Kopierarbeit geleistet werden muss

Übungsaufgabe: Stack (Datenstruktur)

Programmieren Sie eine Klasse `Stack` die eine Integer-Stack verwaltet. Die folgenden Methoden müssen dazu implementiert werden:

```
void push(int x);
int pop();
```

Heap

- unstrukturierter Speicher den man zur Laufzeit vom Betriebssystem anfordern kann
- Speichergröße steht erst zur Laufzeit fest, daher bekommt man vom OS nur einen Pointer

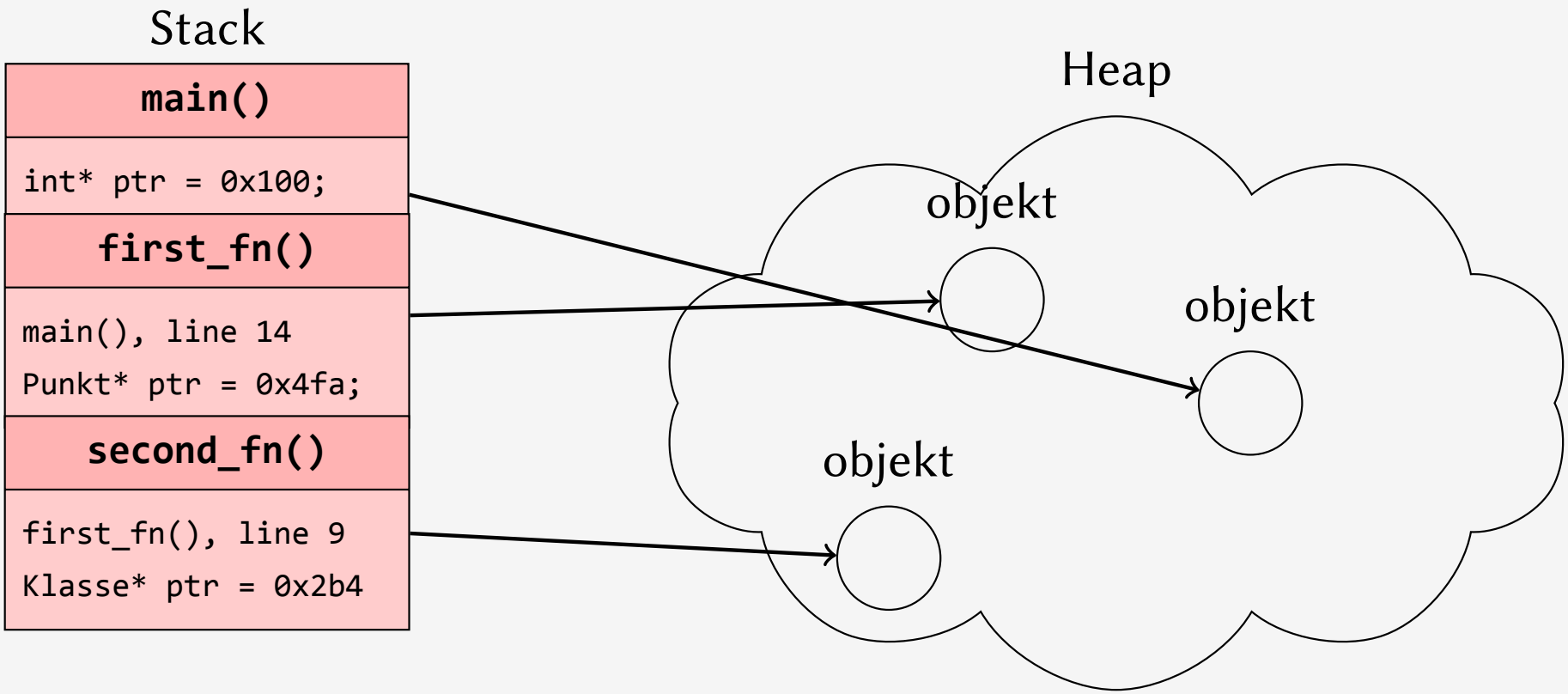
Example (In C)

```c
int* zahl = malloc(sizeof(int));
// ...
free(zahl);
```

Example (In C++)

```cpp
int* zahl = new int;
// ...
delete zahl;
```

Heap Darstellung



Arrays dynamisch auf dem Heap alloziieren

- Arrays mit dynamischer Größe mit `new` auf dem Heap alloziieren

```
cout << "Wie groß soll das Array werden";
cin >> anzahl;
int* dynArray = new int[anzahl];
```

Arrays dynamisch auf dem Heap alloziieren

- Arrays mit dynamischer Größe mit `new` auf dem Heap alloziieren

```
cout << "Wie groß soll das Array werden";
cin >> anzahl;
int* dynArray = new int[anzahl];
```

- Speicher muss wieder mit `delete` gelöscht werden

```
delete[] dynArray;
```

Objekte auf dem Heap

- Objekte auf dem Heap erzeugen mit `new`

```
Klasse* objekt = new Klasse();
```

Objekte auf dem Heap

- Objekte auf dem Heap erzeugen mit `new`

```
Klasse* objekt = new Klasse();
```

- Zugriffe auf Attribute und Methoden würde normalerweise so aussehen:

```
(*objekt).attribut = 23;
(*objekt).vergroessern();
```

Objekte auf dem Heap

- Objekte auf dem Heap erzeugen mit `new`

```
Klasse* objekt = new Klasse();
```

- Zugriffe auf Attribute und Methoden würde normalerweise so aussehen:

```
(*objekt).attribut = 23;
(*objekt).vergroessern();
```

- Vereinfachung über den -> Operator

```
objekt->attribut = 23;
objekt->vergroessern();
```

Objekte auf dem Heap

Example (Stack)

```
#include <iostream>
#include "Rechteck.h"

using namespace std;

int main(void) {
    Rechteck quadrat;

    quadrat.hoehe  = 23;
    quadrat.breite = 23;

    quadrat.vergroessern(10);

    cout << quadrat.flaecheninhalt();
}
```

Objekte auf dem Heap

Example (Stack)

```
#include <iostream>
#include "Rechteck.h"

using namespace std;

int main(void) {
    Rechteck quadrat;

    quadrat.hoehe  = 23;
    quadrat.breite = 23;

    quadrat.vergroessern(10);

    cout << quadrat.flaecheninhalt();
}
```

Example (Heap)

```
#include <iostream>
#include "Rechteck.h"

using namespace std;

int main(void) {
    Rechteck* quadrat = new Rechteck();

    quadrat->hoehe  = 23;
    quadrat->breite = 23;

    quadrat->vergroessern(10);

    cout << quadrat->flaecheninhalt();

    delete quadrat;
}
```

Memory Leaks sind Bugs

Example (Memory-Leak)

```
Klasse* objekt = new Klasse();
//...
Klasse* objekt2 = new Klasse();
objekt = objekt2;
delete objekt;
```

Vor- und Nachteile des Heaps

Vorteile

- Speicher kann zur Laufzeit allokiert werden
- Dynamisch allokierter Speicher bleibt allokiert bis er mit `delete` gelöscht wird
- Heap ist größer als der Stack, daher könnten dort große Array oder Objekte allokiert werdne

Nachteile

- Memory-Leaks

Zusammenfassung

- Pointer/Referenzen sind Datentypen die Speicheradressen speichern
- Unterschied zwischen Pointer und Referenzen
- Funktions- und Methodenaufrufe über Call-by-Value, Call-By-Reference und Pointer-Based Call-by-Reference
- Arrays und was diese mit Pointern gemeinsam haben
- Grundsatzliche Funktionsweise der Speicherverwaltung
- Unterschied Stack und Heap