Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Jakob Vogel

Computer-Aided Medical Procedures
Technische Universität München

Administratives

- Sprech**punkt** (Montag, 13:30, Raum 0999)
  - Falls Sie *begründet* erst später kommen können, melden Sie sich bitte per Mail oder mündlich während der ZÜ an!
  - Spätestes Ende ist um 14:30 Uhr.
- Lösungsblätter und Übung **ergänzen** sich bisweilen.
- Beachten Sie die Anleitungen für *Visual Studio* und *Xcode* auf der Website!

`algods@mailnavab.in.tum.de`

`http://campar.in.tum.de/Chair/TeachingWs12AlgoDs`

Warum C **und** C++?

- C ist die Grundlage, aber einige Konzepte fehlen, sind umständlich oder fehleranfällig.
- C++ ist eine (von mehreren) Erweiterungen von C und korrigiert diese Probleme (weitergehend).
- Alle Konzepte von C funktionieren auch in C++, aber manches vermeidet man besser!

- C++ (oder Vergleichbares wie *Java* oder *Objective C*) ist heute der Standard für „normale“ Programme!

Wie geht es weiter?

- Wir haben mit dem nächsten Aufgabenblatt die (für diese Veranstaltung) wichtigsten Konzepte behandelt.
- Für volle Beherrschung der Sprache fehlt aber noch einiges!
- Beispielcode kann daher „neue“ Konstrukte beinhalten...

- Fragen Sie, oder konsultieren Sie Referenzen!

`http://cppreference.com/`

`http://cplusplus.com/`

Zahldarstellung

- Eine Zahl wird dargestellt als Summe von Zweierpotenzen:

$$23 = 1 + 2 + 4 + 16 = 2^0 + 2^1 + 2^2 + 2^4$$

- In Binärdarstellung betrachten wir nur die Koeffizienten:

$$23_{10} = 10111_2$$

- Hexadezimal-Schreibweise ist praktisch:

0_{16}	0000_2	$\cdots$	$\cdots$	C_{16}	1100_2
1_{16}	0001_2	$\cdots$	$\cdots$	D_{16}	1101_2
2_{16}	0010_2	$\cdots$	$\cdots$	E_{16}	1110_2
3_{16}	0011_2	$\cdots$	$\cdots$	F_{16}	1111_2

1 Hexadezimal- fasst 4 Binär-Koeffizienten zusammen!

$$FF_{16} = 1111\ 1111_2 = 255_{10}$$

Speicherung von Ganzzahlen (I)

- Einzelne Koeffizienten werden in *Bits* (0/1) gespeichert, 8 Bits sind ein *Byte*.
- Ganze Zahlen werden (bei 64-bit-Systemen) in 1, 2, 4 oder 8 Bytes gespeichert.
- Bei mehr als einem Byte kann man sich über *Big*- und *Little-Endian*-Speicherung streiten!

$$12345_{10} = 3039_{16} = 0011\ 0000\ 0011\ 1001_2$$

Unterschied im Byte-weise organisierten Speicher:

30 39 vs. 39 30

Speicherung von Ganzzahlen (II)

- Negative Zahlen werden im Zweierkomplement dargestellt:

$$-n_2 \rightsquigarrow (\overline{n_2}) + 1$$

- „Nur“ ein Bitmuster, Interpretation erfolgt im Kontext!
- Daher unterscheidet man `signed` und `unsigned` Typen!
- Typennamen wie `int` sind nur bedingt verläßlich!
  - `int8_t`, `uint8_t`
  - `int16_t`, `uint16_t`
  - ...

Speicherung von Fließkommazahlen

- Speicherung als Vorzeichen, Mantisse und Exponent.
- Berechnungen sind sehr teuer!.
- Es existieren mindestens die Typen `float` (32 Bit) und `double` (64 Bit).

Primitive Datentypen

- Ganzzahlen (`int`, usw.)
- Fließkommazahlen (`float`, `double`)

- Wahrheitswerte sind auch Ganzzahlen!

$$\texttt{wert == true} \iff \texttt{wert != 0} \iff \texttt{wert}$$
$$\texttt{wert == false} \iff \texttt{wert == 0} \iff \texttt{!wert}$$

- Buchstaben sind Ganzzahlen, abgebildet auf Schriftzeichen nach Tabelle (ASCII, Unicode, etc.)!

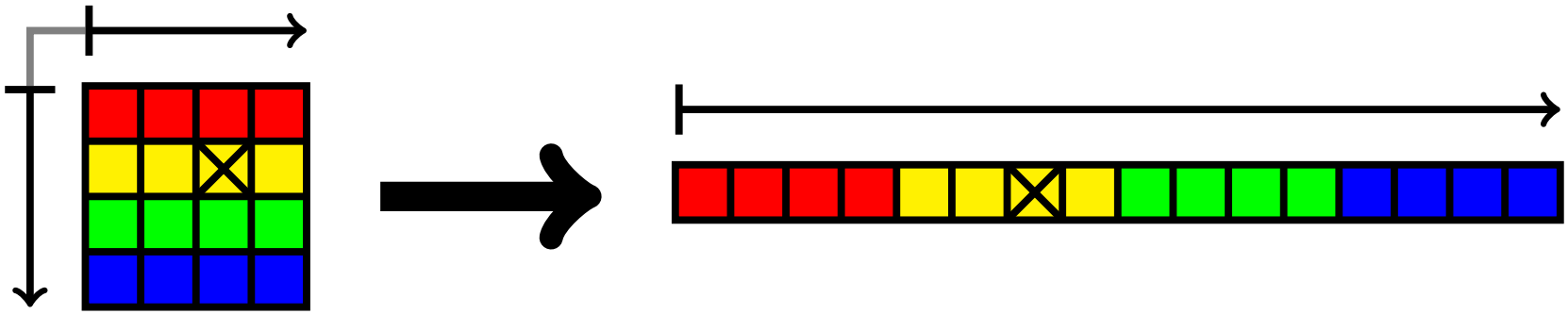
Bitweise Operationen (I)

- Können **bei Ganzzahlen** teure arithmetische Funktionen ersetzen.
- Bitshift links/rechts entspricht Multiplikation mit/Teilung durch 2:

$$\begin{aligned} 6_{10} &= 0110_2 = 2^2 + 2^1 && \mid \cdot 2 \\ 12_{10} &= 1100_2 = 2^2 \cdot 2 + 2^1 \cdot 2 = 2^3 + 2^2 \end{aligned}$$

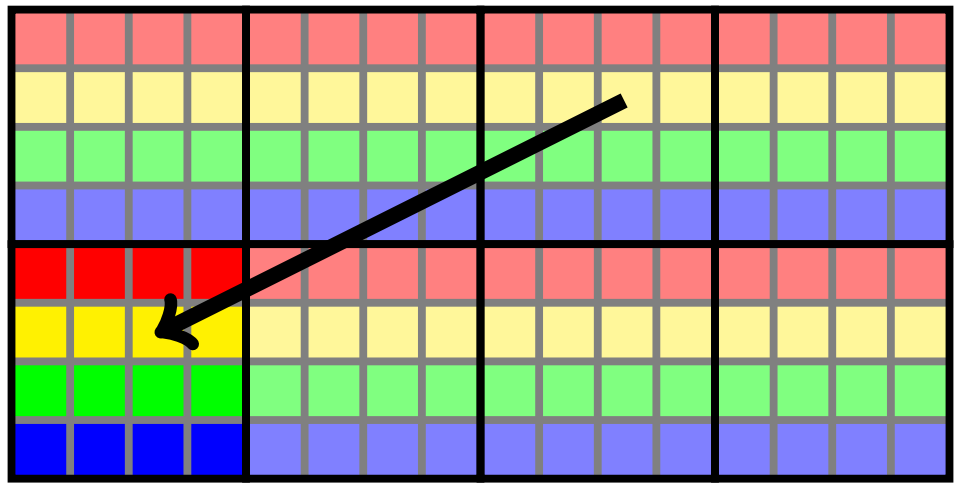
- UND mit $2^n - 1$ ergibt Teilungsrest bei Division durch 2^n.
- Derartiges ist sehr schnell und einfach in Hardware realisierbar!

Texturen für 3D-Grafik (I)



- Wir beginnen zu zählen bei 0!
- Speicher ist ein-dimensional organisiert, also liegt der Farbwert bei $y \cdot \mathit{breite} + x$ im Speicher.
- Das markierte Pixel (2/1) wird abgebildet auf Index $1 \cdot 4 + 2 = 6$, schnell berechnet als `(y << 2) + x`!

Texturen für 3D-Grafik (II)



- Bei gekachelten Texturen kann mittels UND mit $3 = 4 - 1$ in das Originalbild abgebildet werden.
- Daher wählt man bis heute gerne Zweier-Potenzen als Texturgröße!

Bitweise Operationen (II)

- Auch komplexere Operationen sind durch bitweise Operationen realisierbar!
- `http://graphics.stanford.edu/~seander/bithacks.html`


Code

Da war noch was...

- Diese Schleife terminiert **nicht**:

```
for (unsigned int i = 50; i >= 0; --i)
    std::cout << "i␣=␣" << i << std::endl;
```

- `unsigned`-Typen sind per Konstruktion **immer** ≥ 0
- Bei Berechnungen kann es zum *Überlauf* kommen!

Array

- Sozusagen Kette von n Variablen
- Indizierung mit $\mathbf{0} \leq \mathbf{i} < \mathbf{n}$

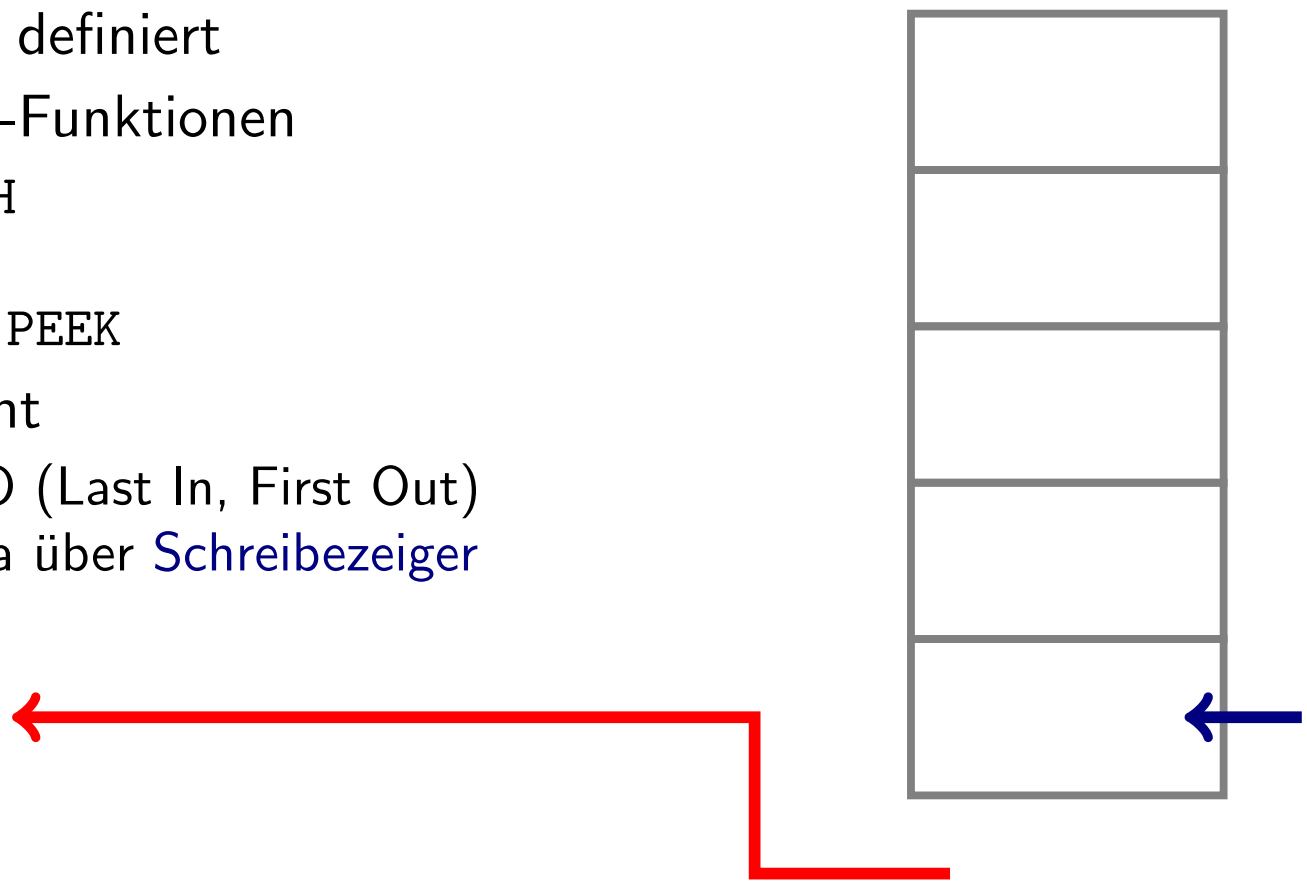
- Schon drei „Schreibweisen“ bisher:
  - ```
    // Fixe Anzahl von Elementen
    int array[5];
    ```
  - ```
    // Variable Anzahl, aber Speichermanagement
    int* array = new int[n]; /* ... */ delete[] array;
    ```
  - ```
    // Variable Anzahl, aber std::vector
    #include <vector>
    std::vector<int> array(n);
    ```
    ```


Code

Stack

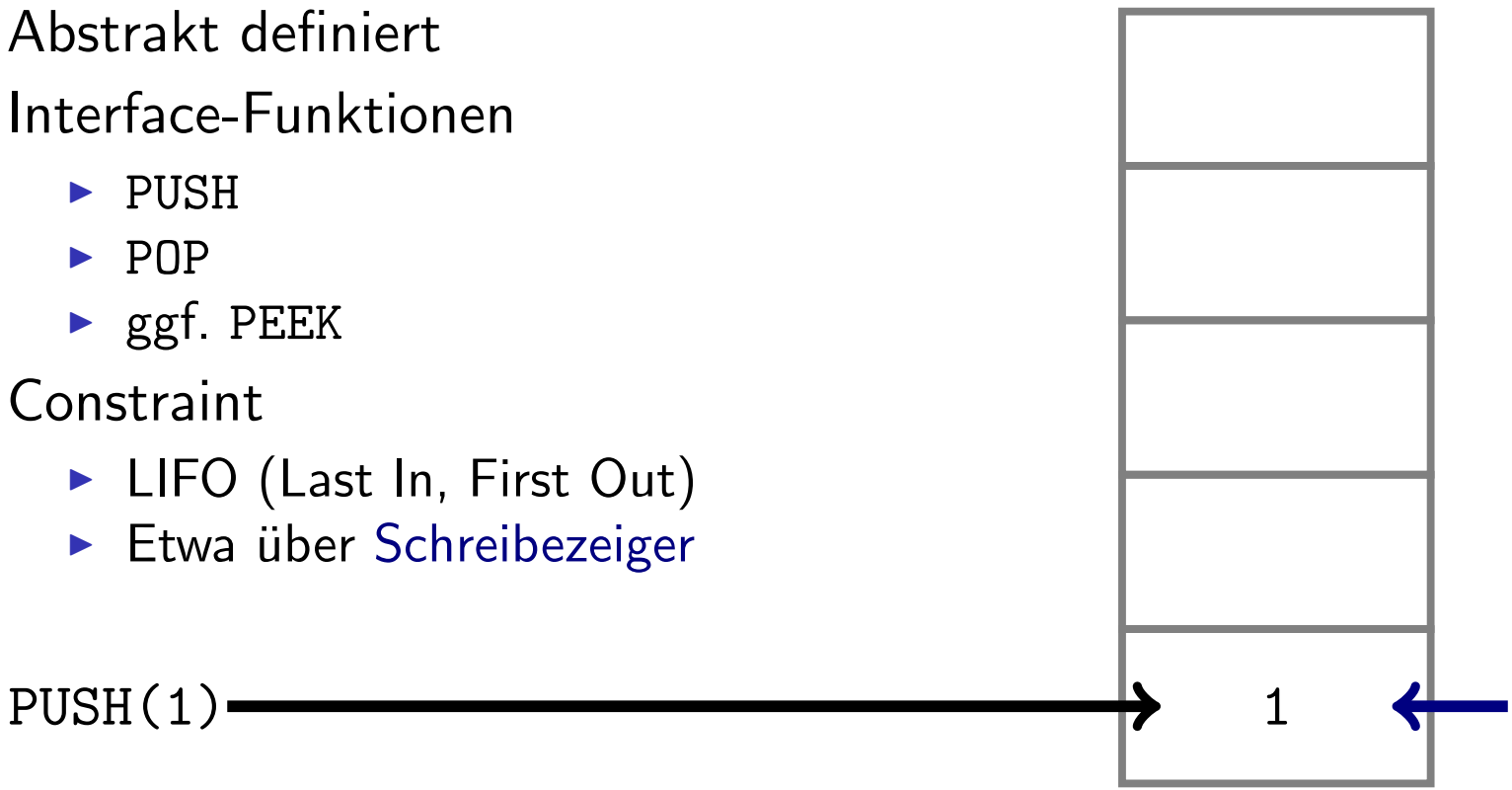
- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger

`POP = ?`



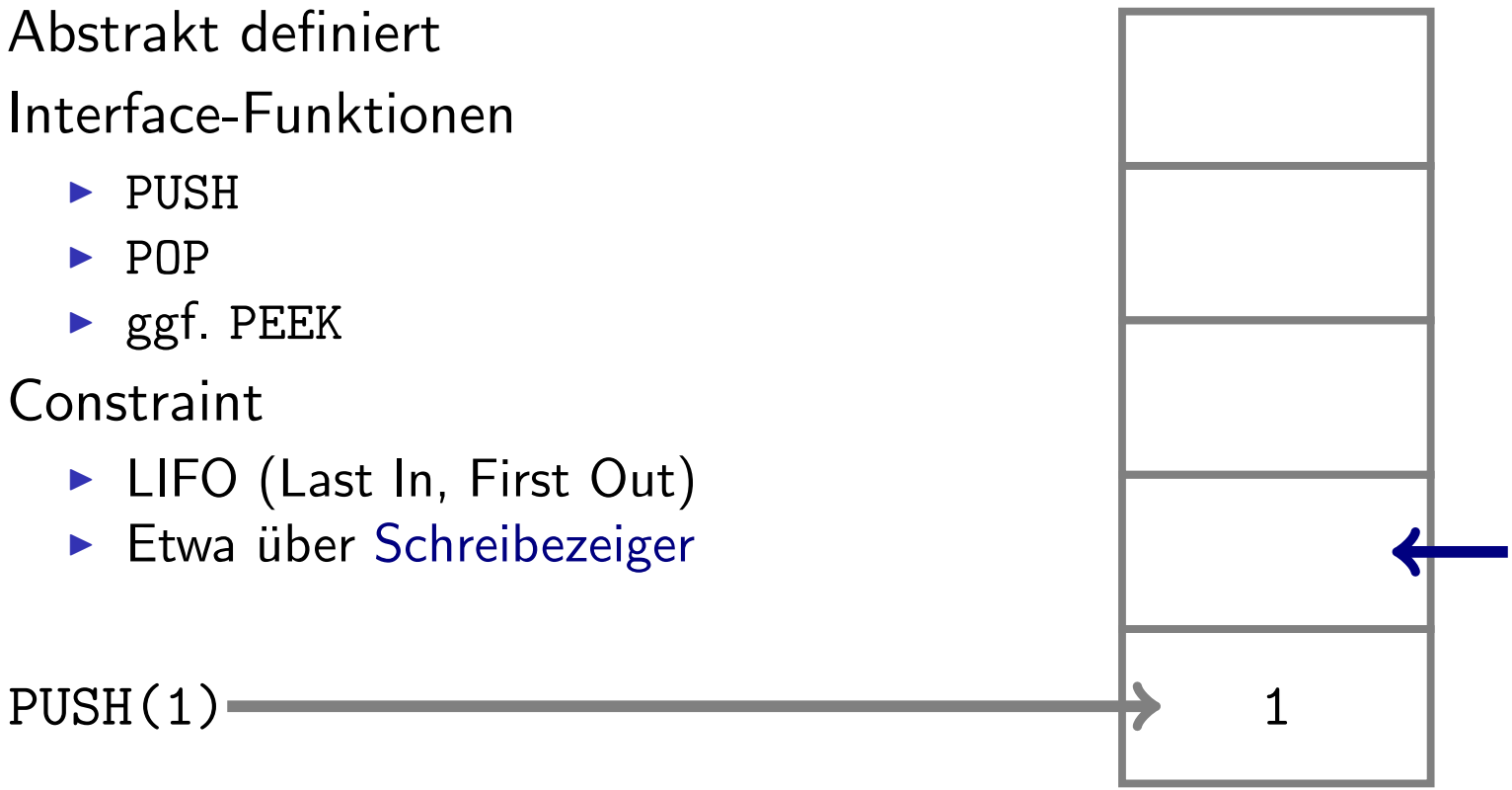
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



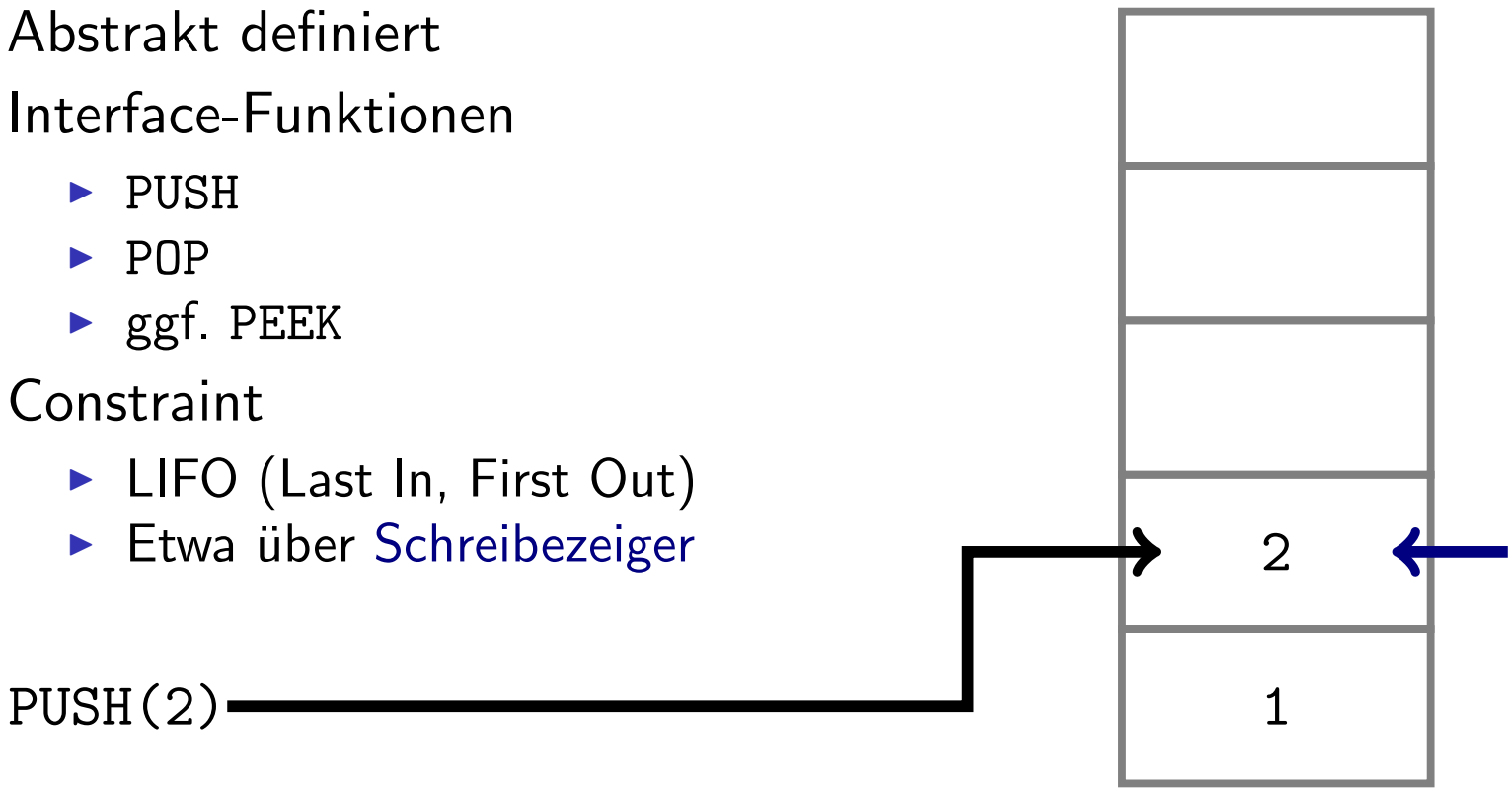
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



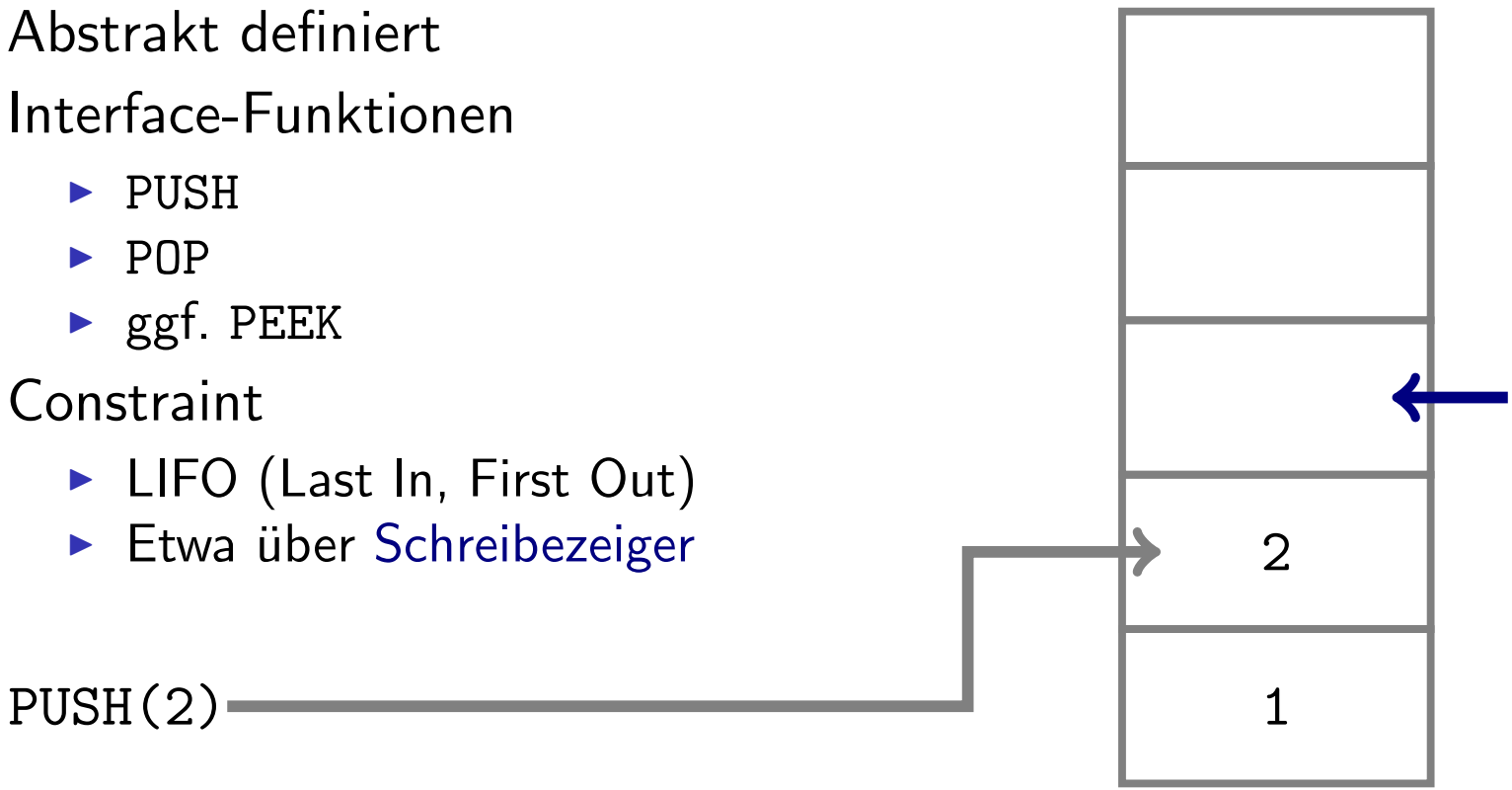
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



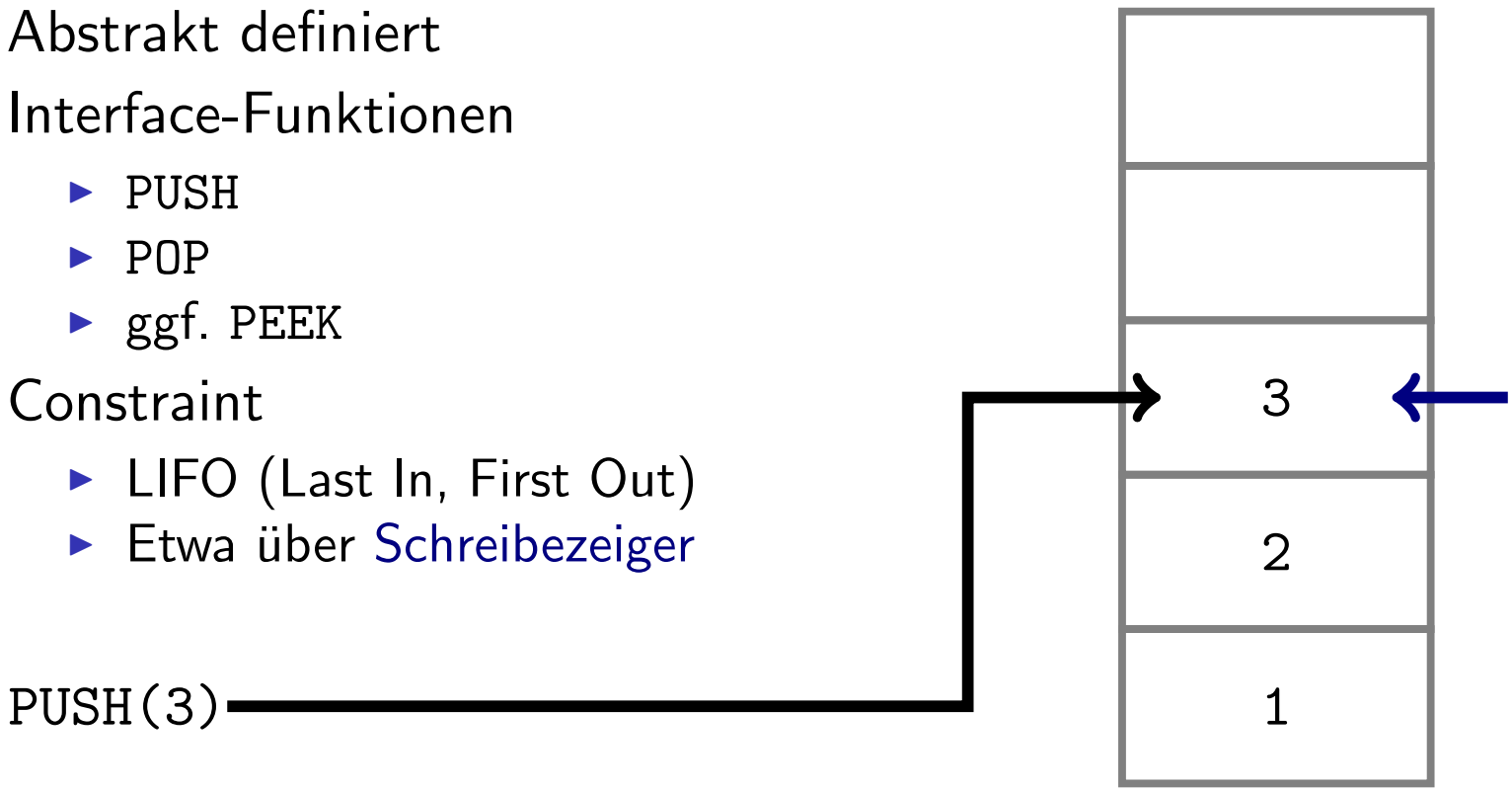
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



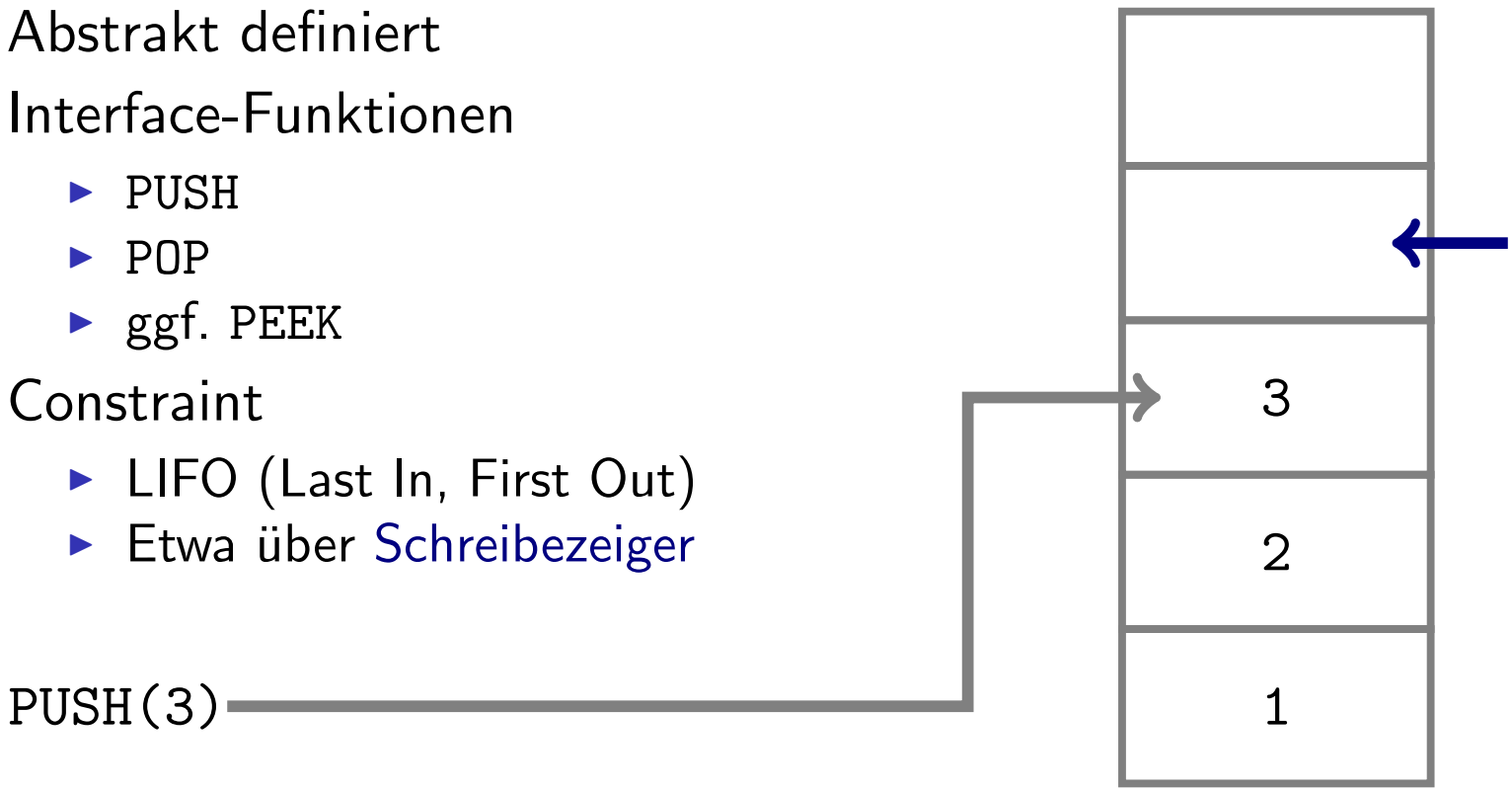
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



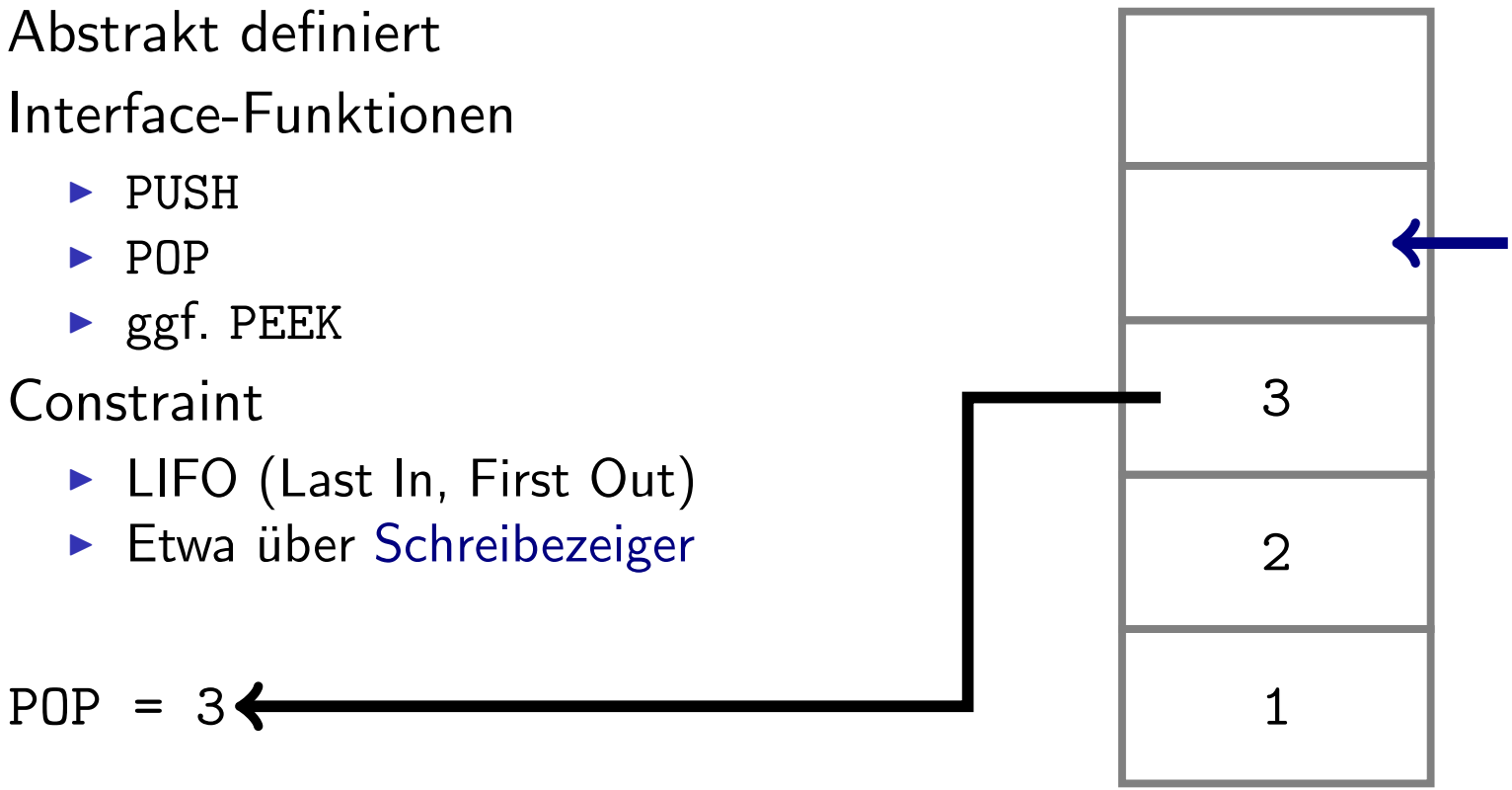
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



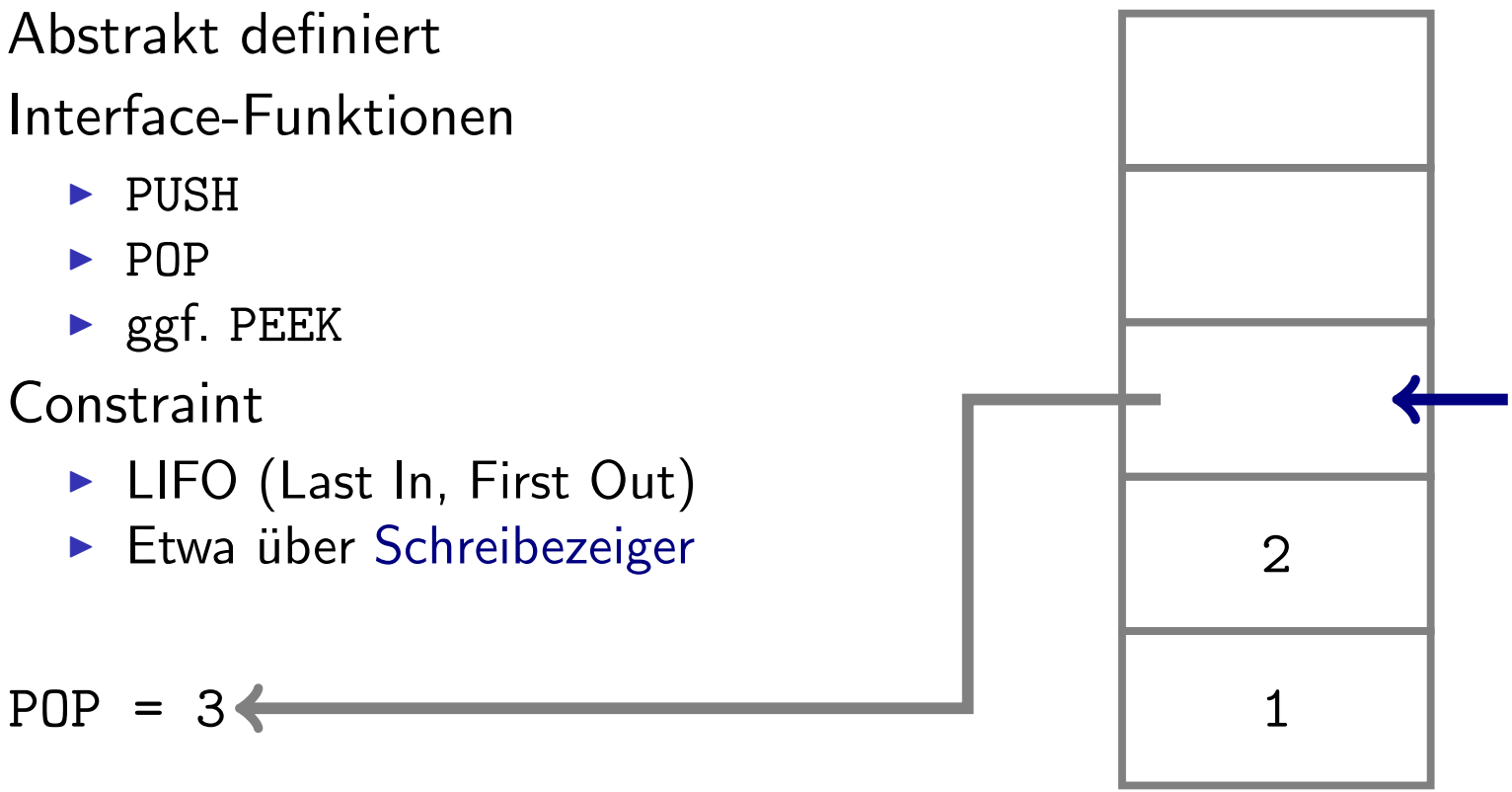
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



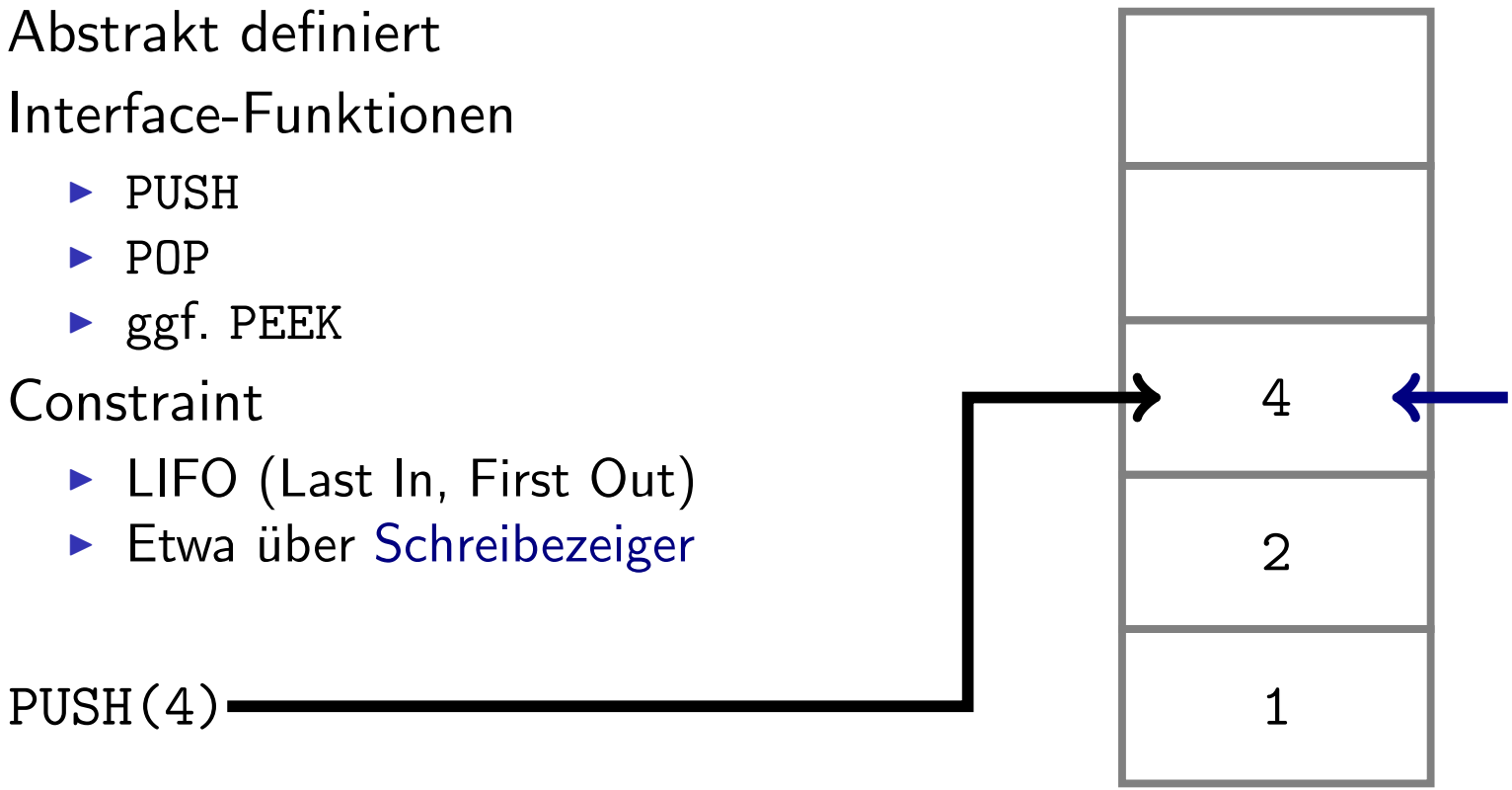
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



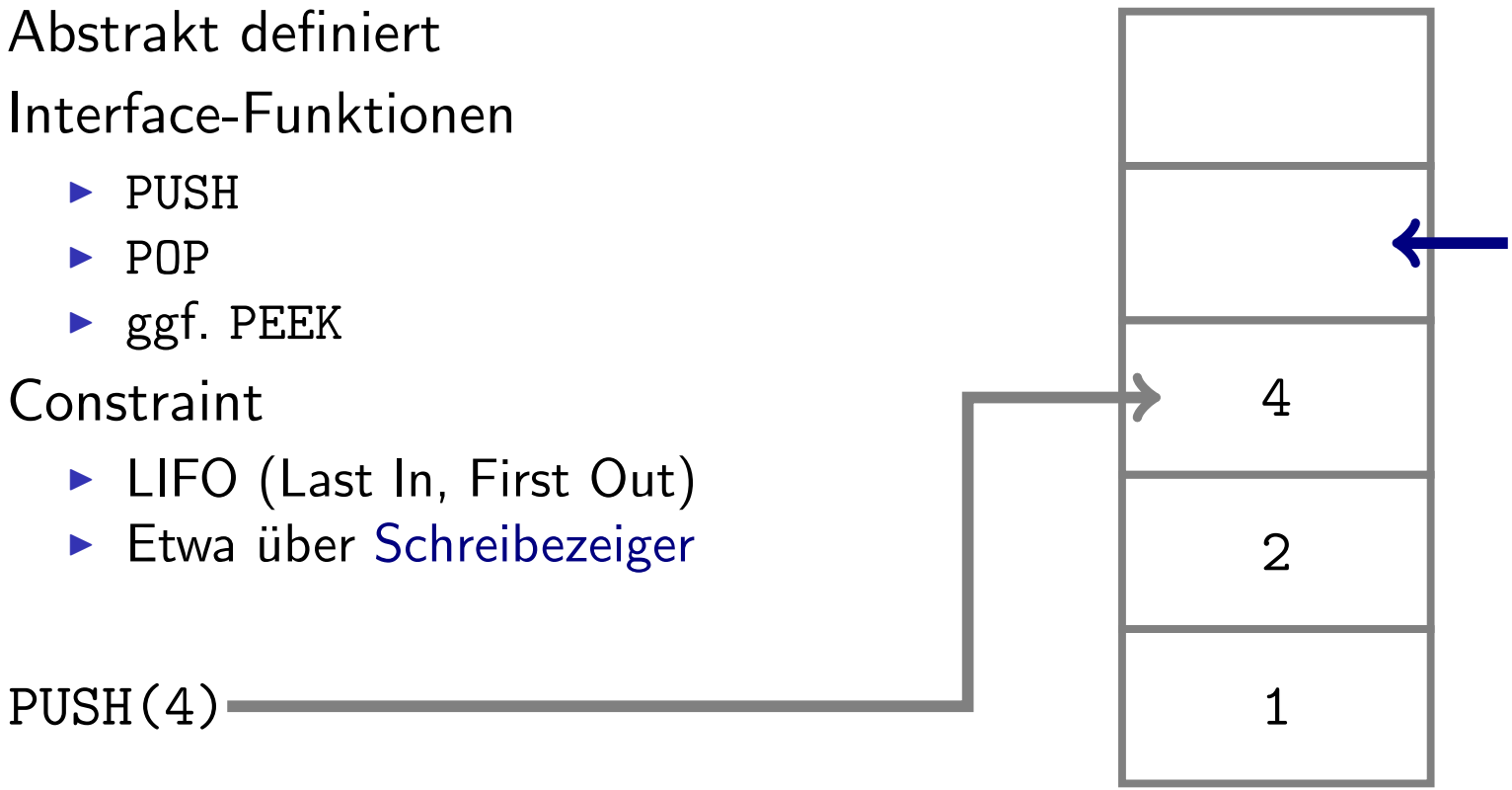
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



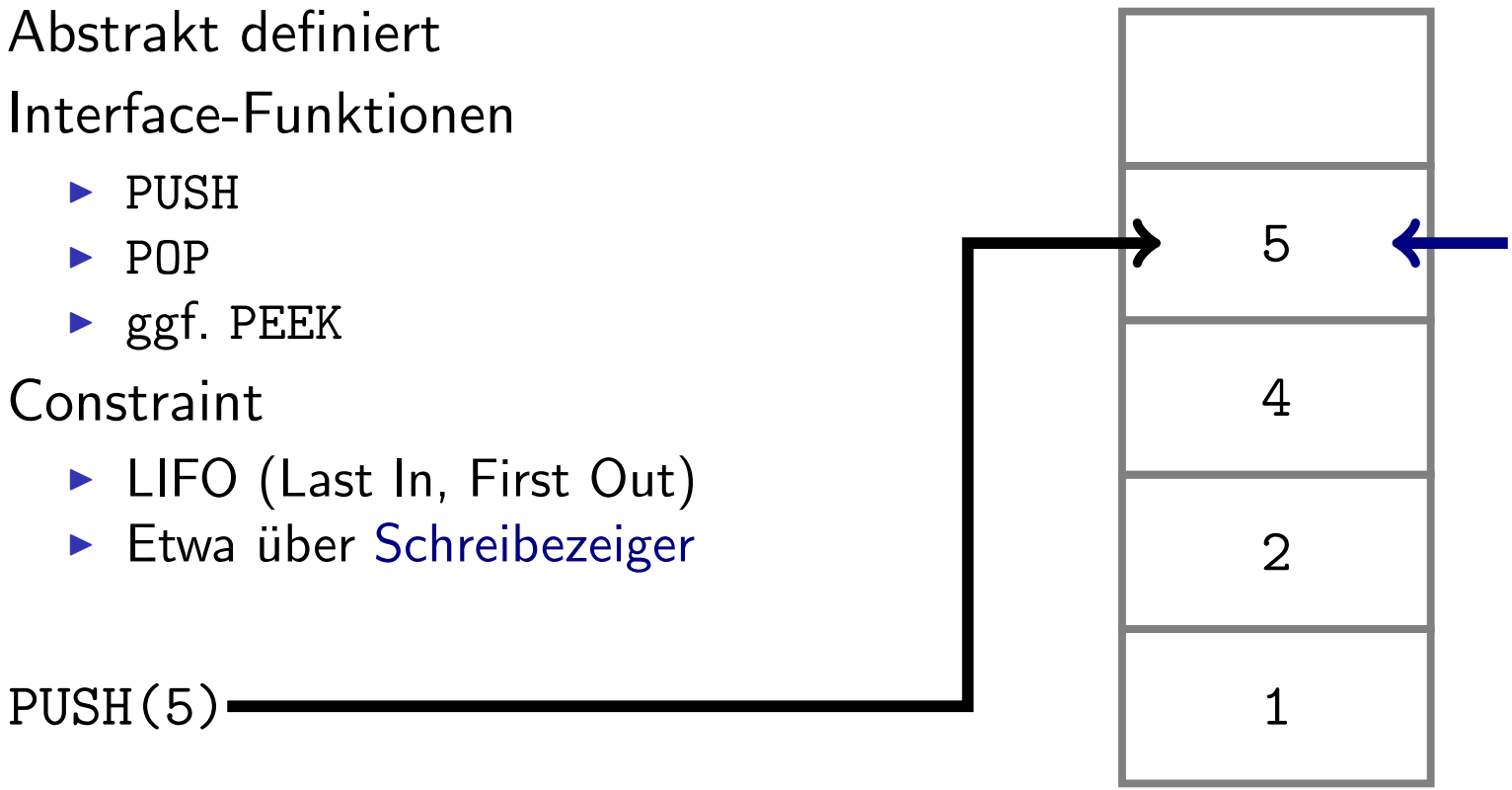
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



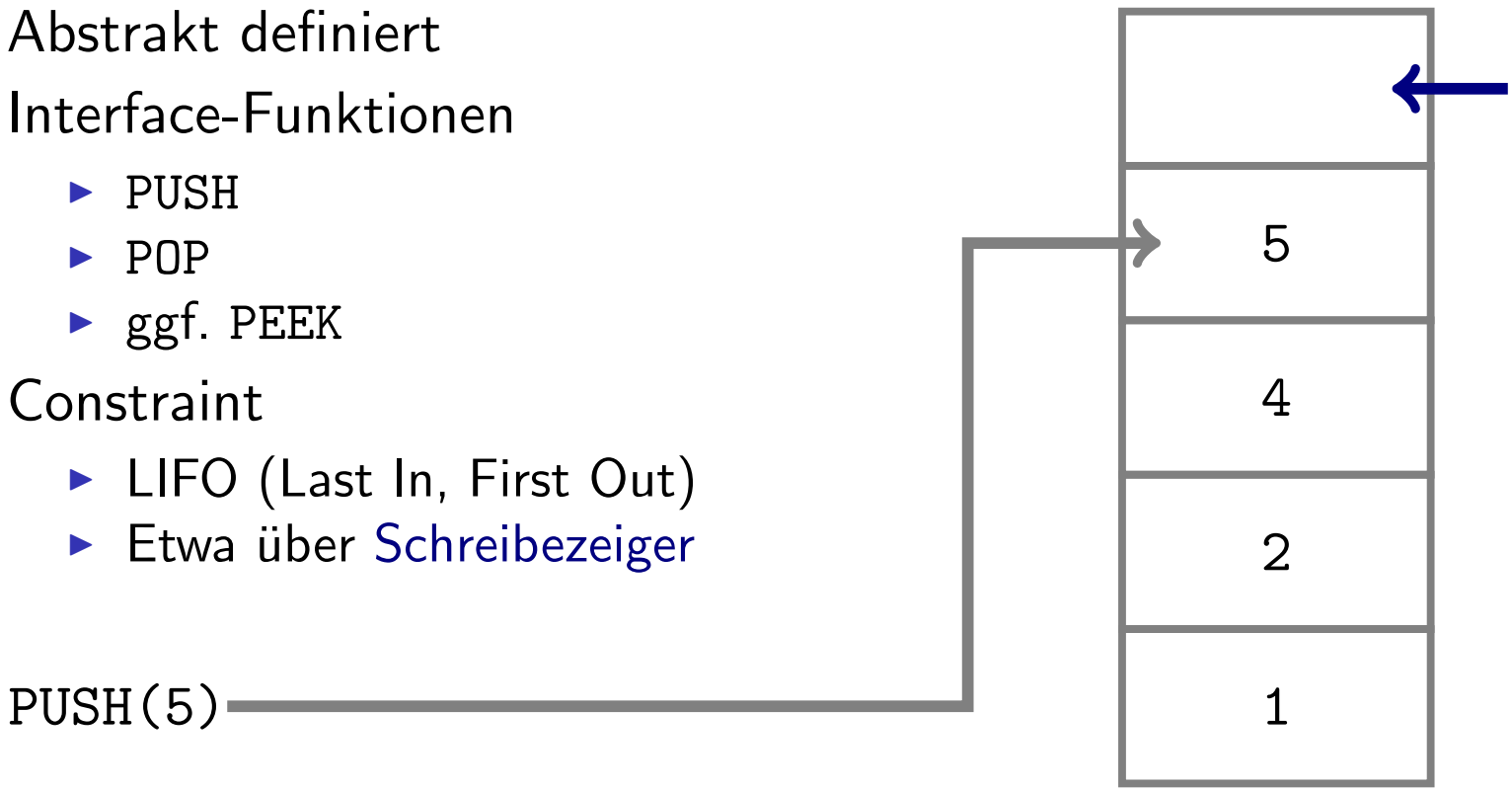
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



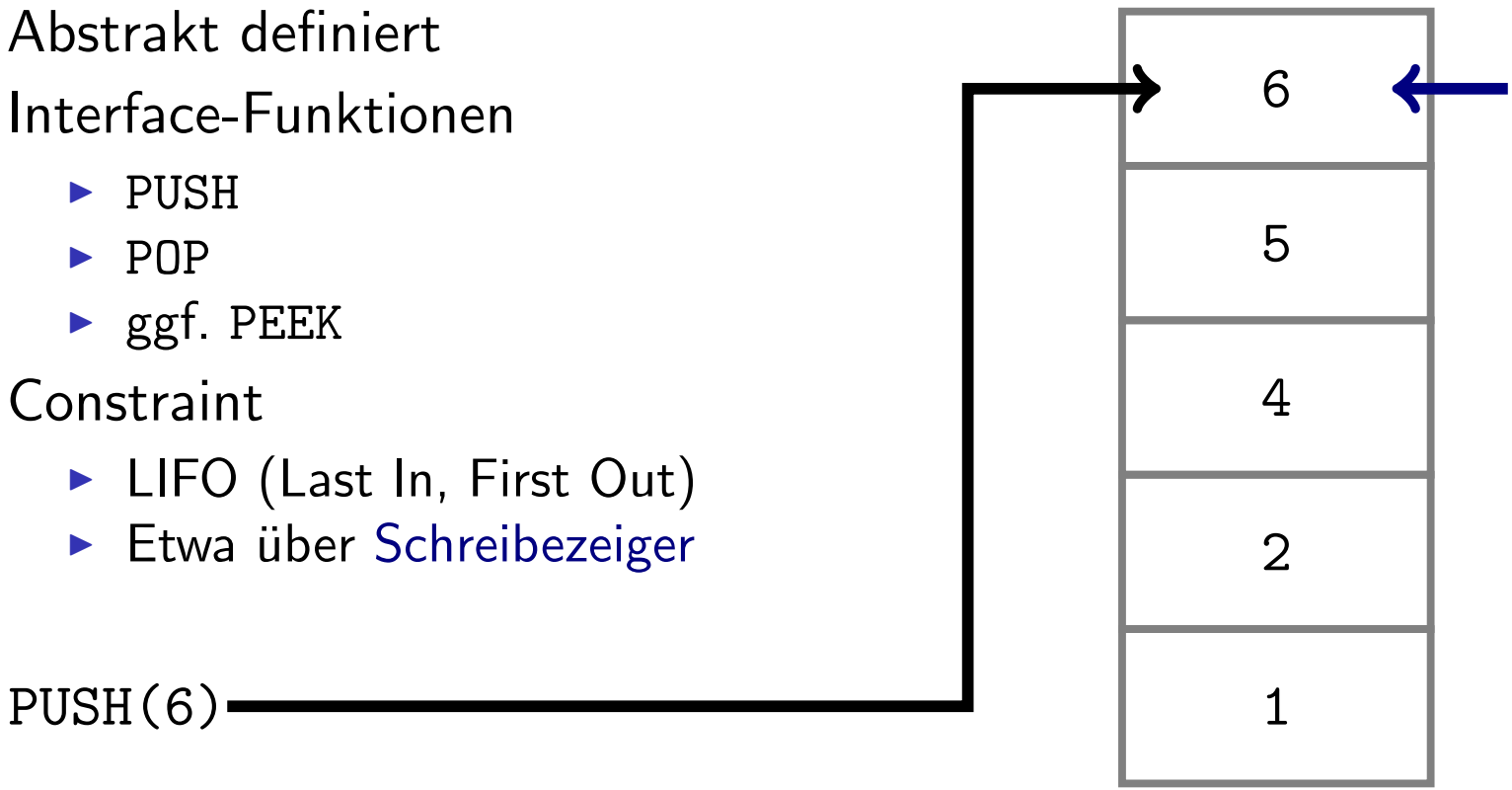
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



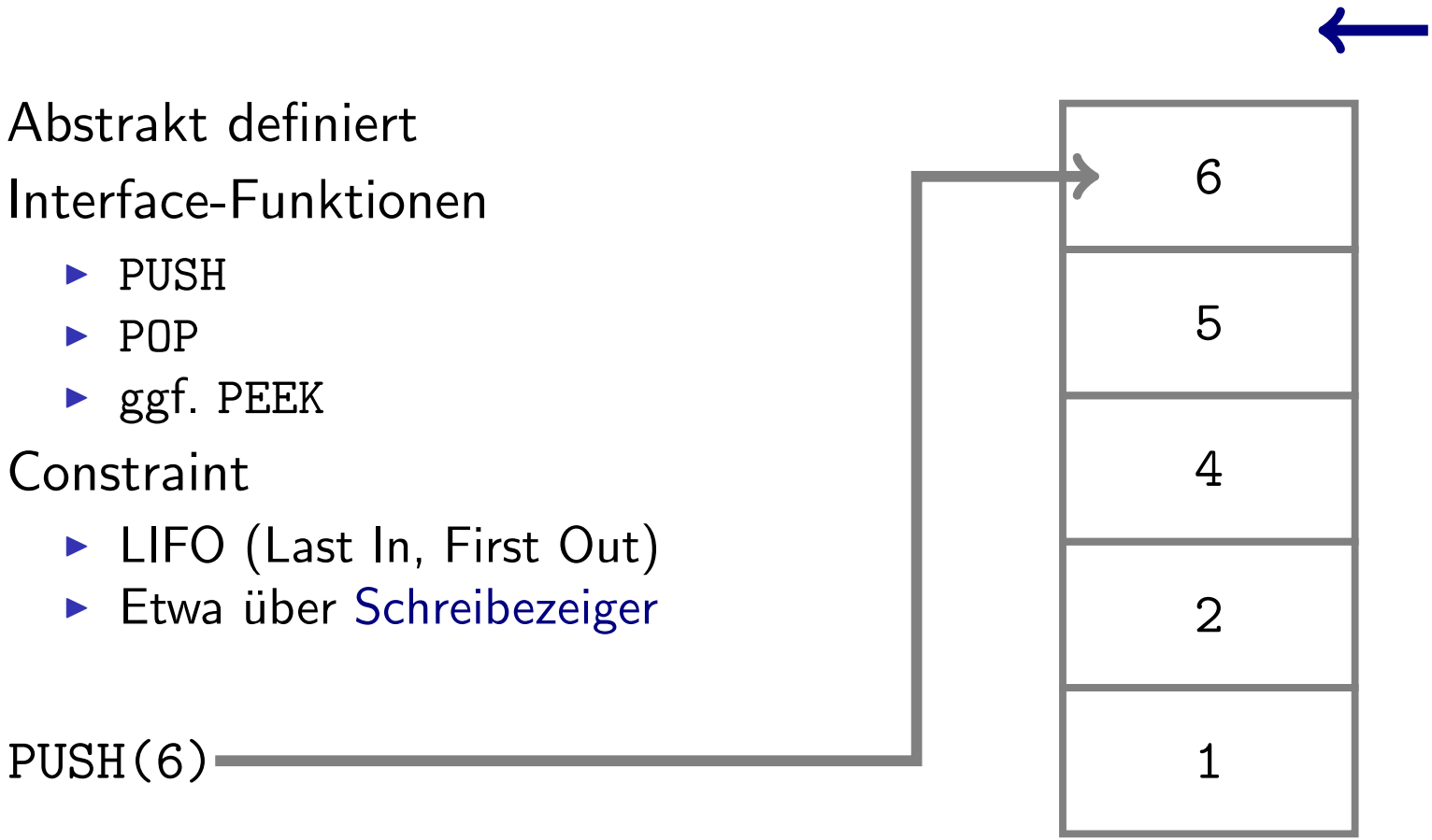
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



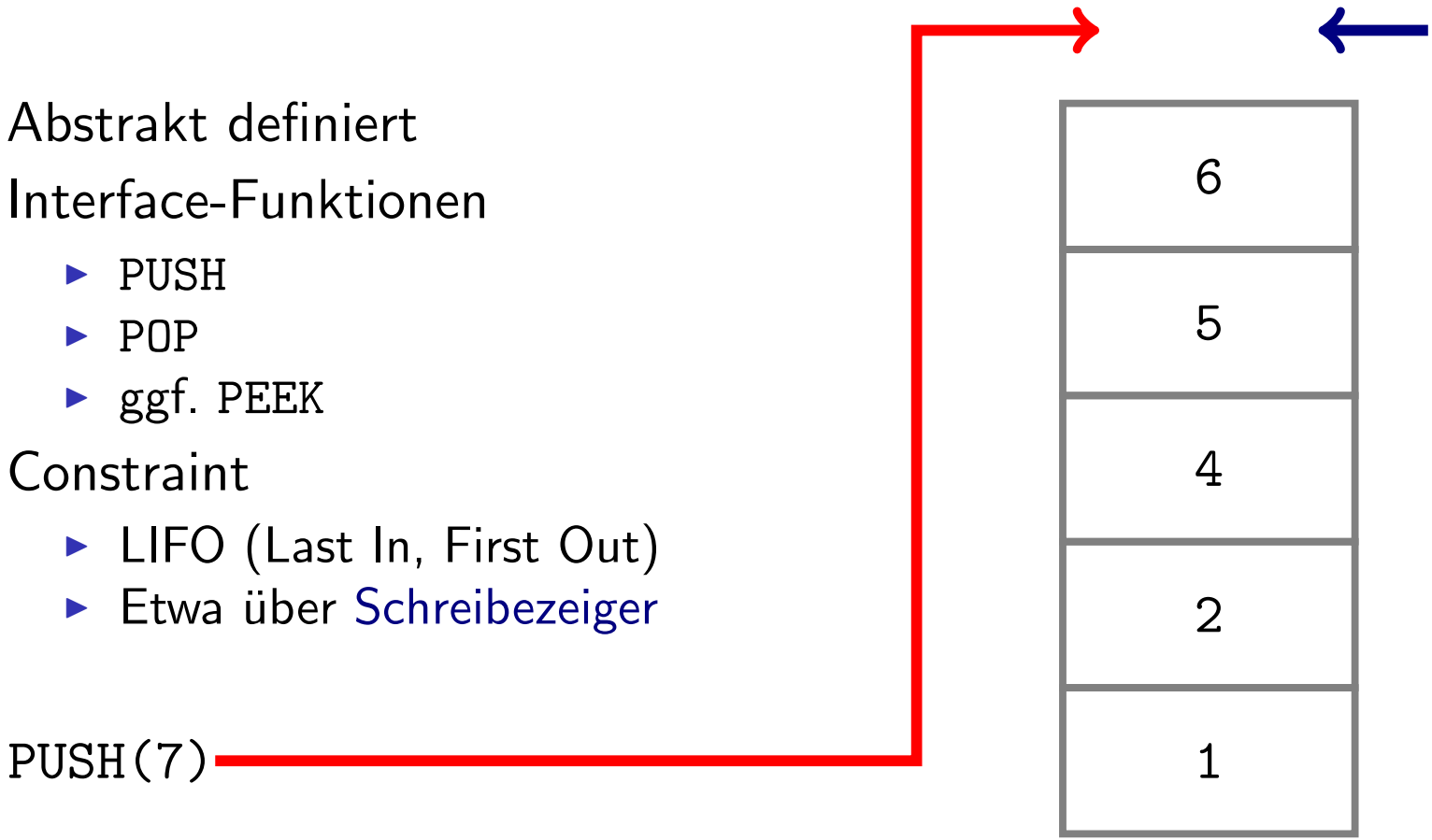
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



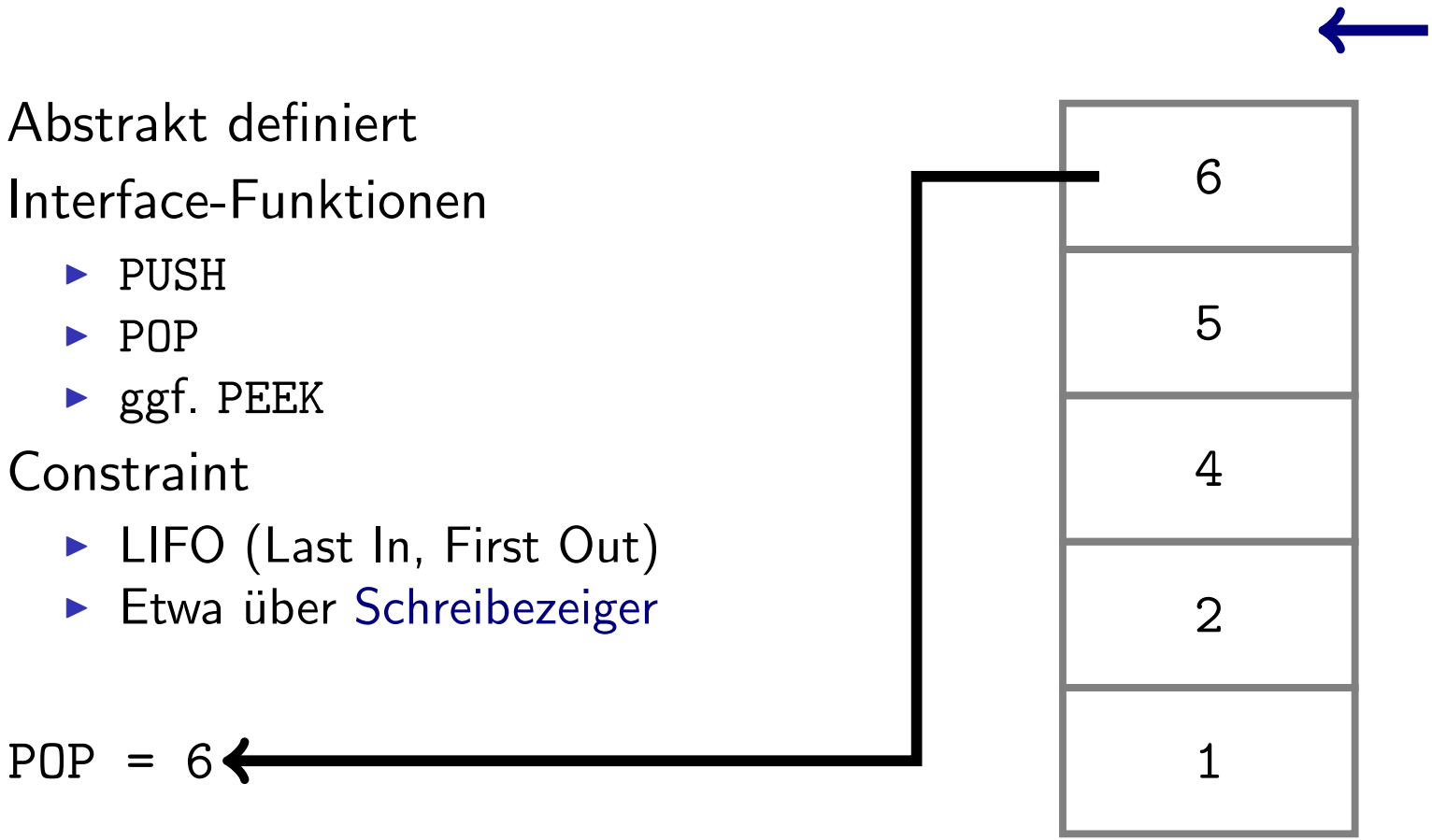
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



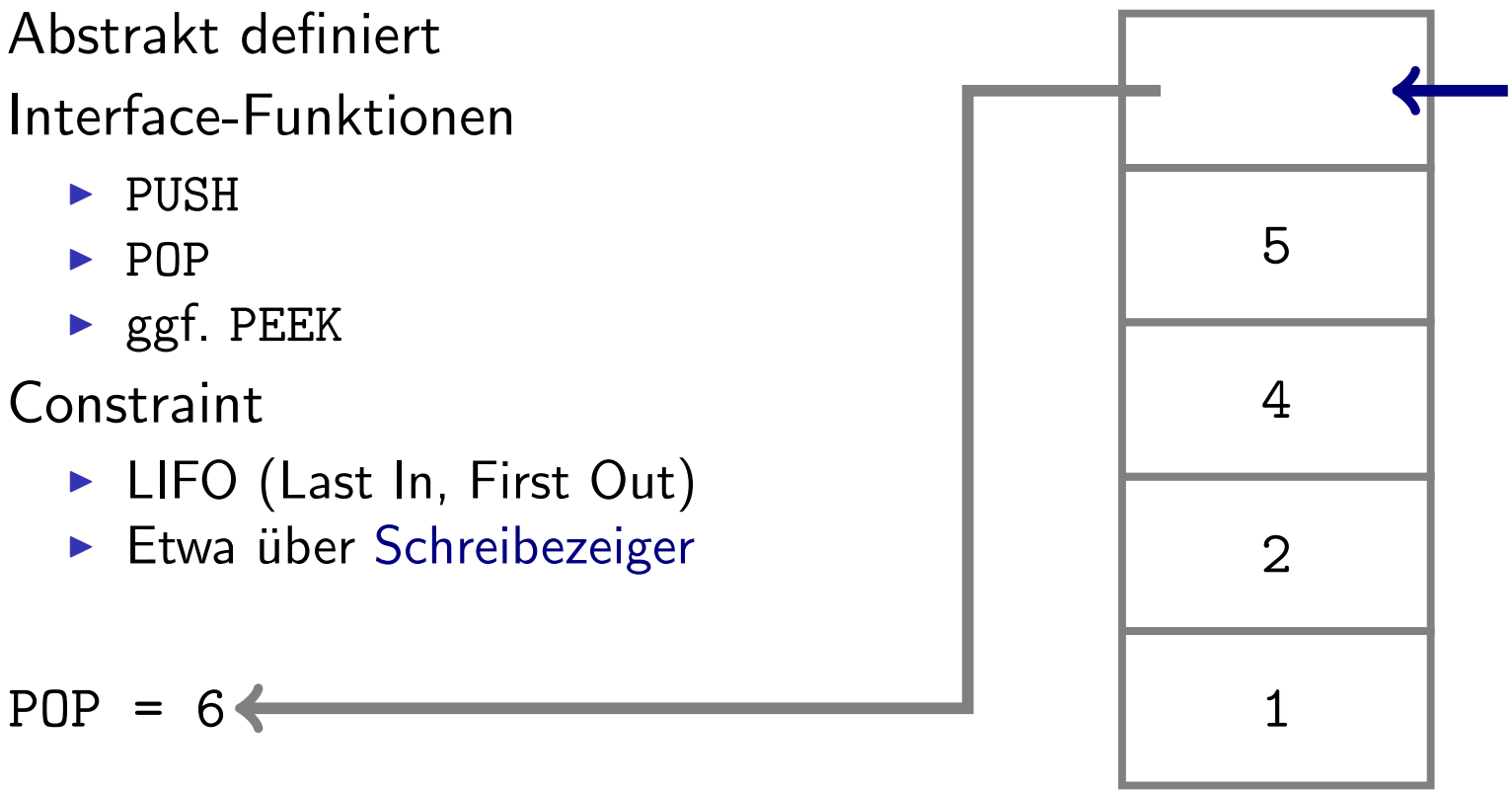
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



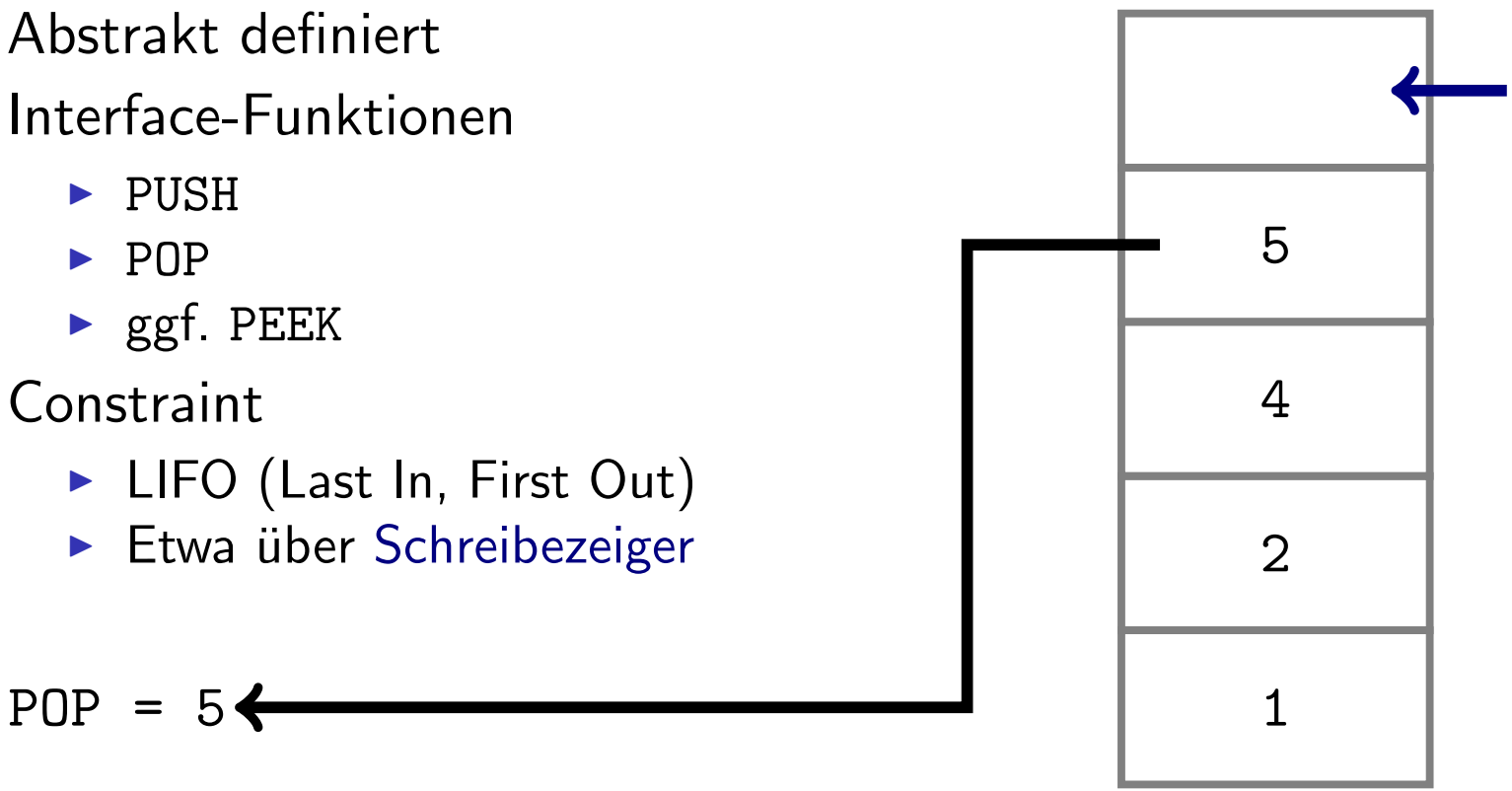
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



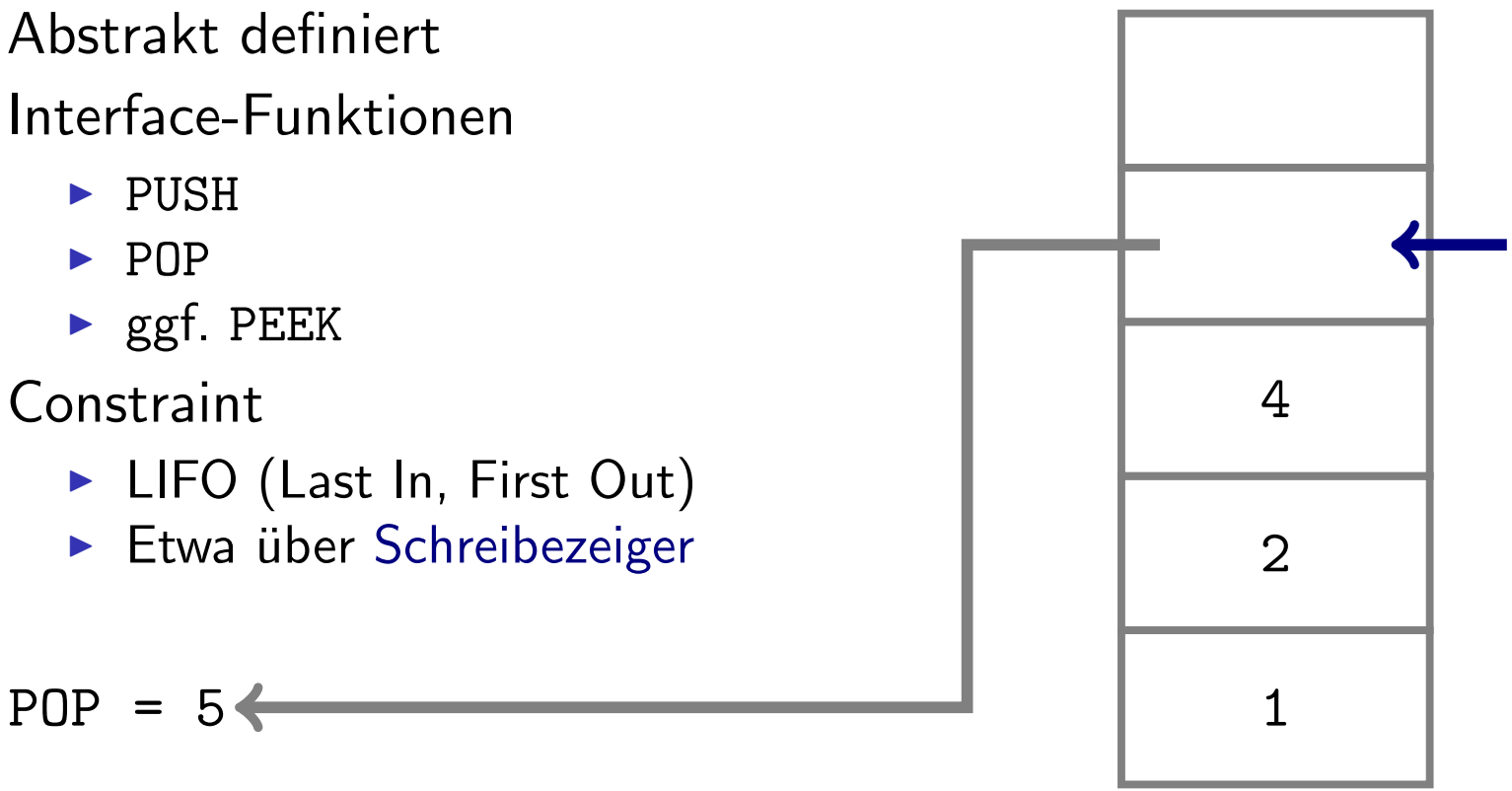
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



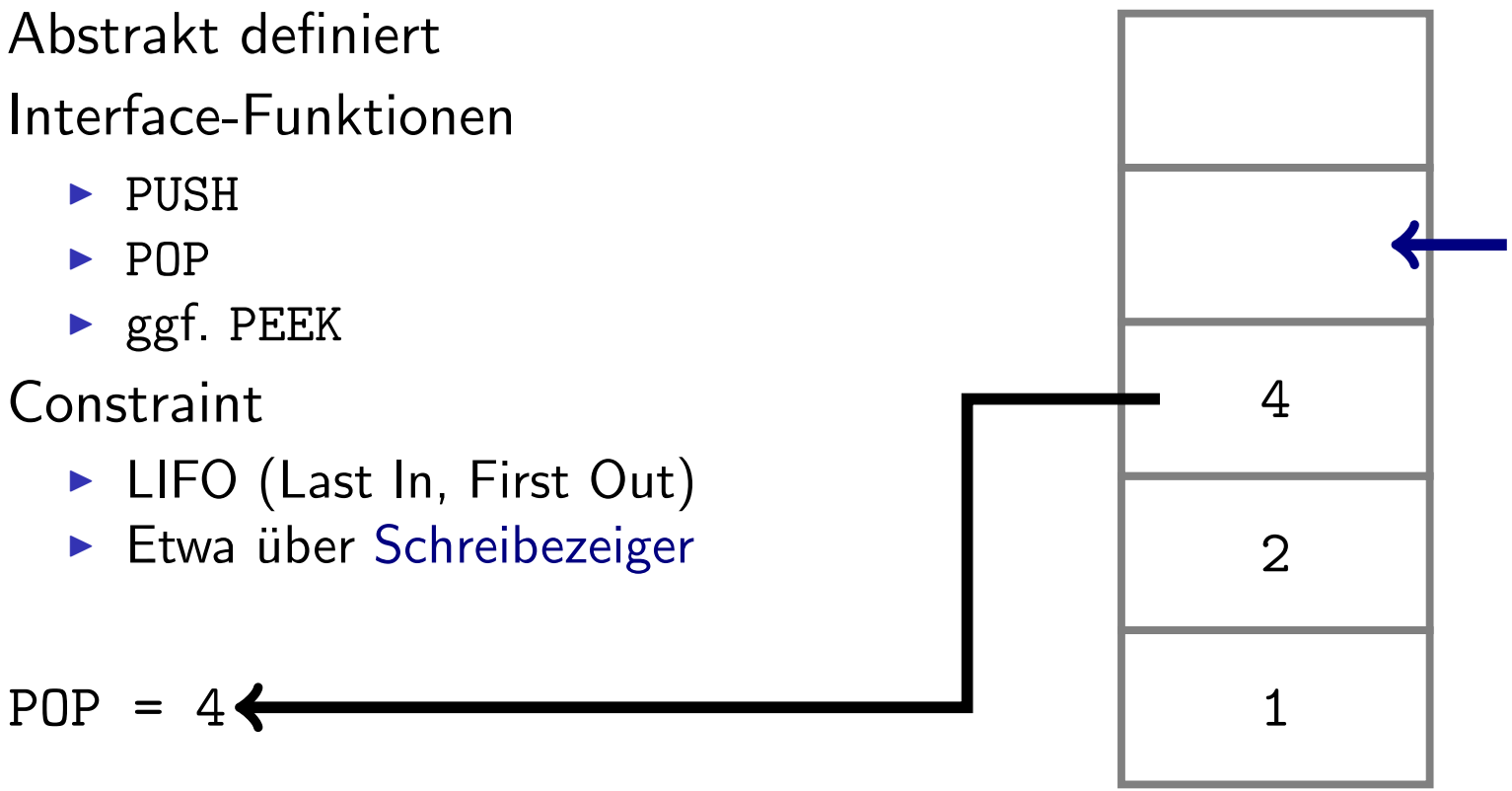
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



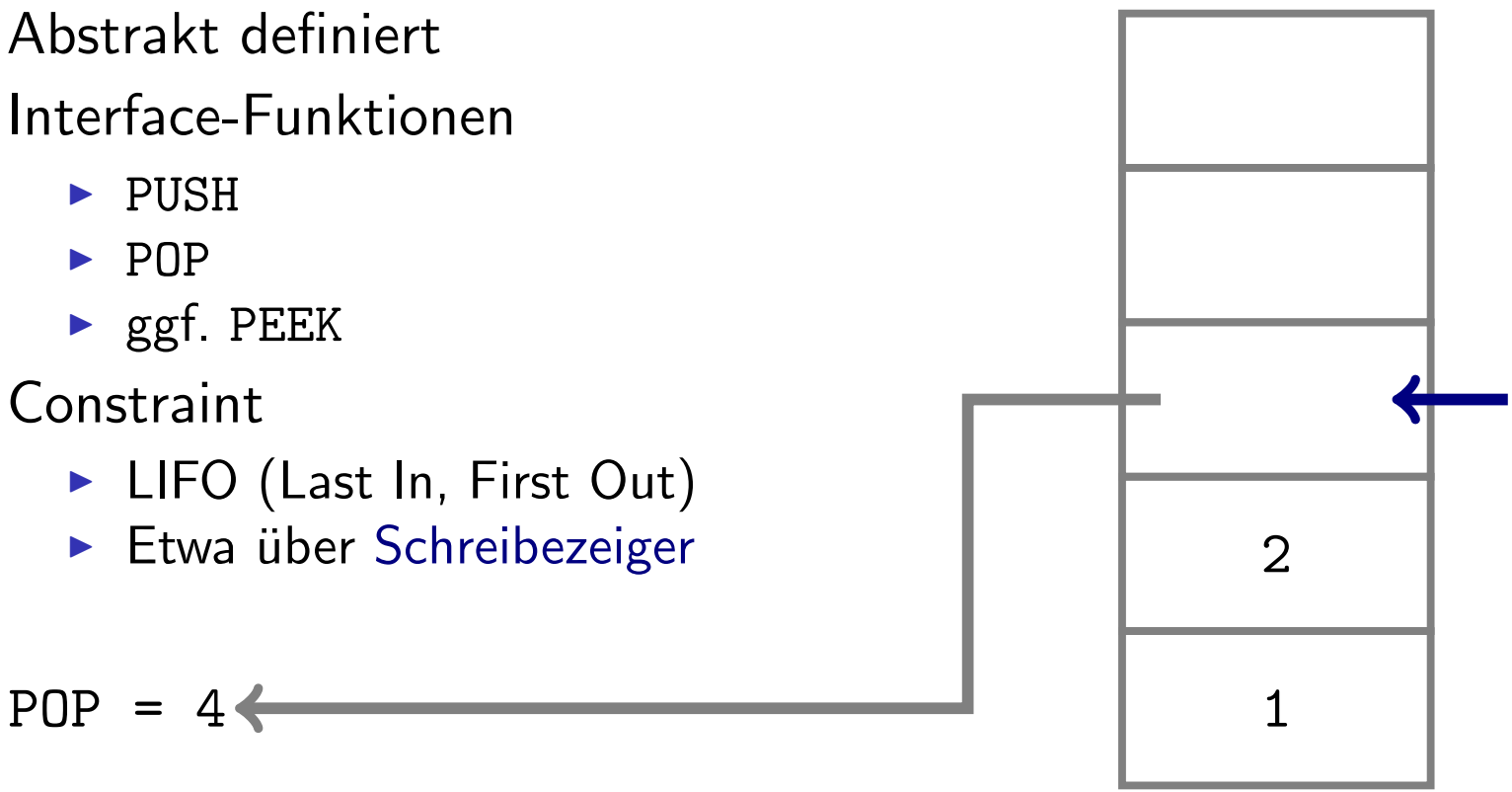
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



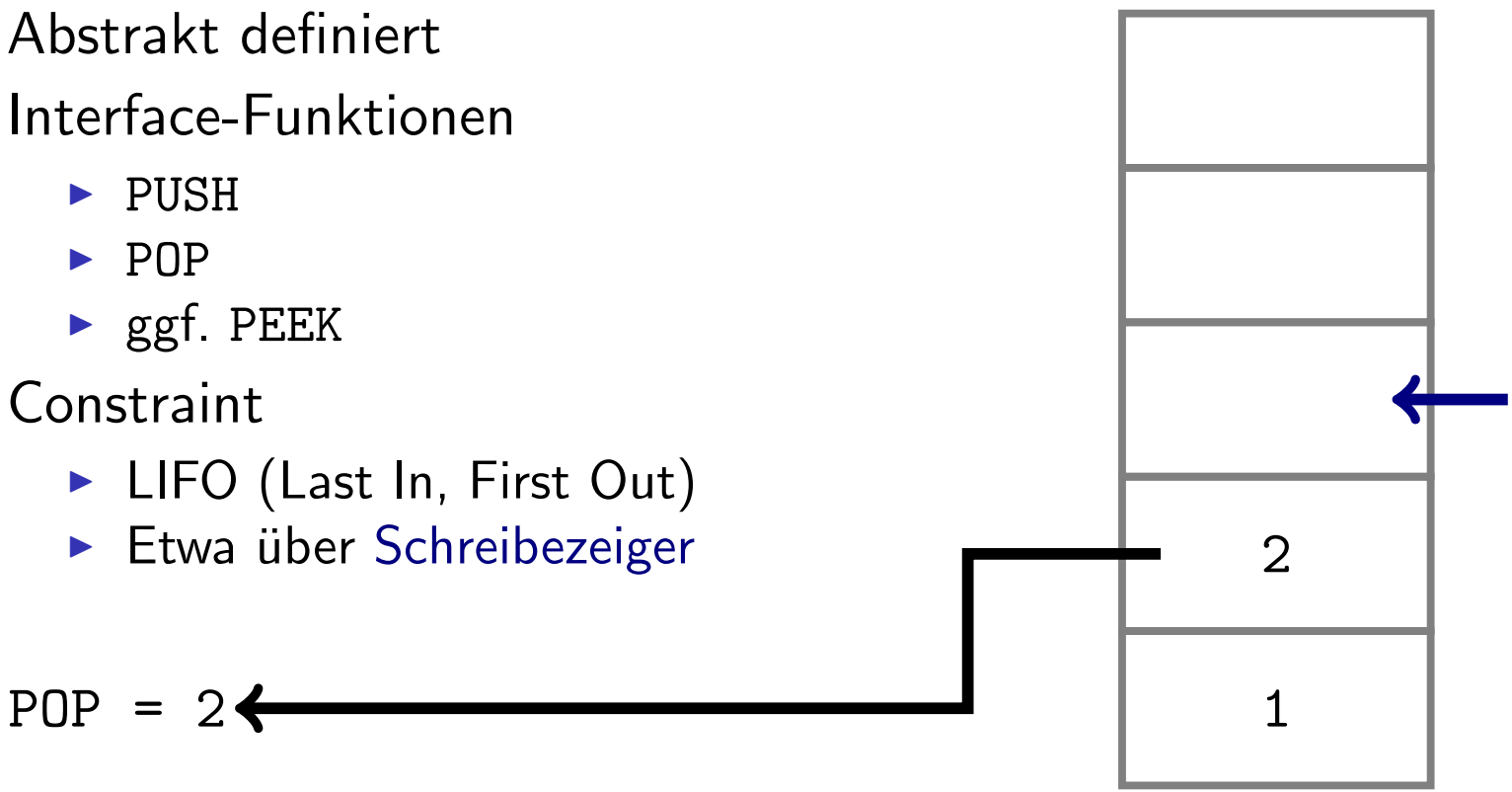
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



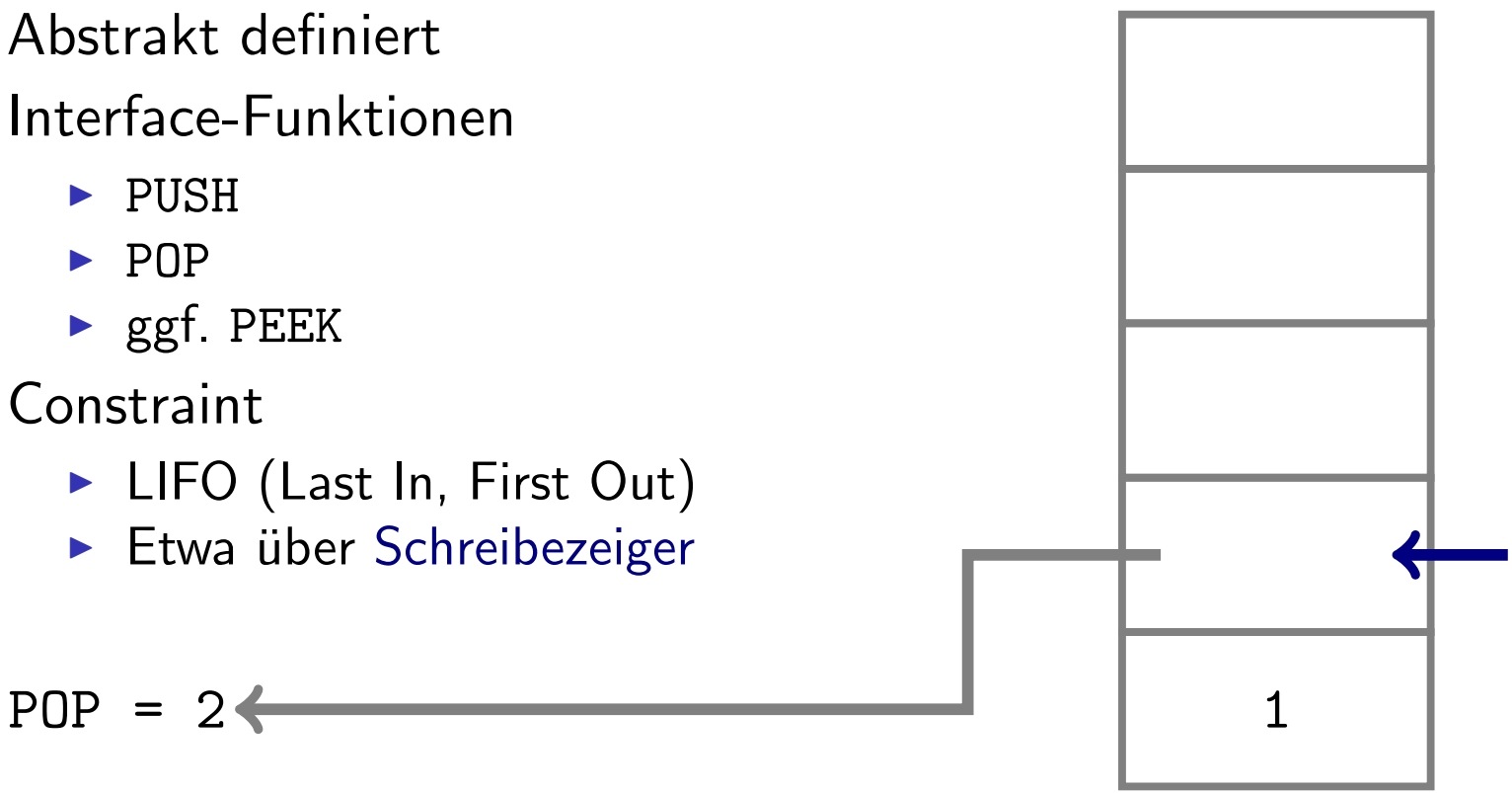
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



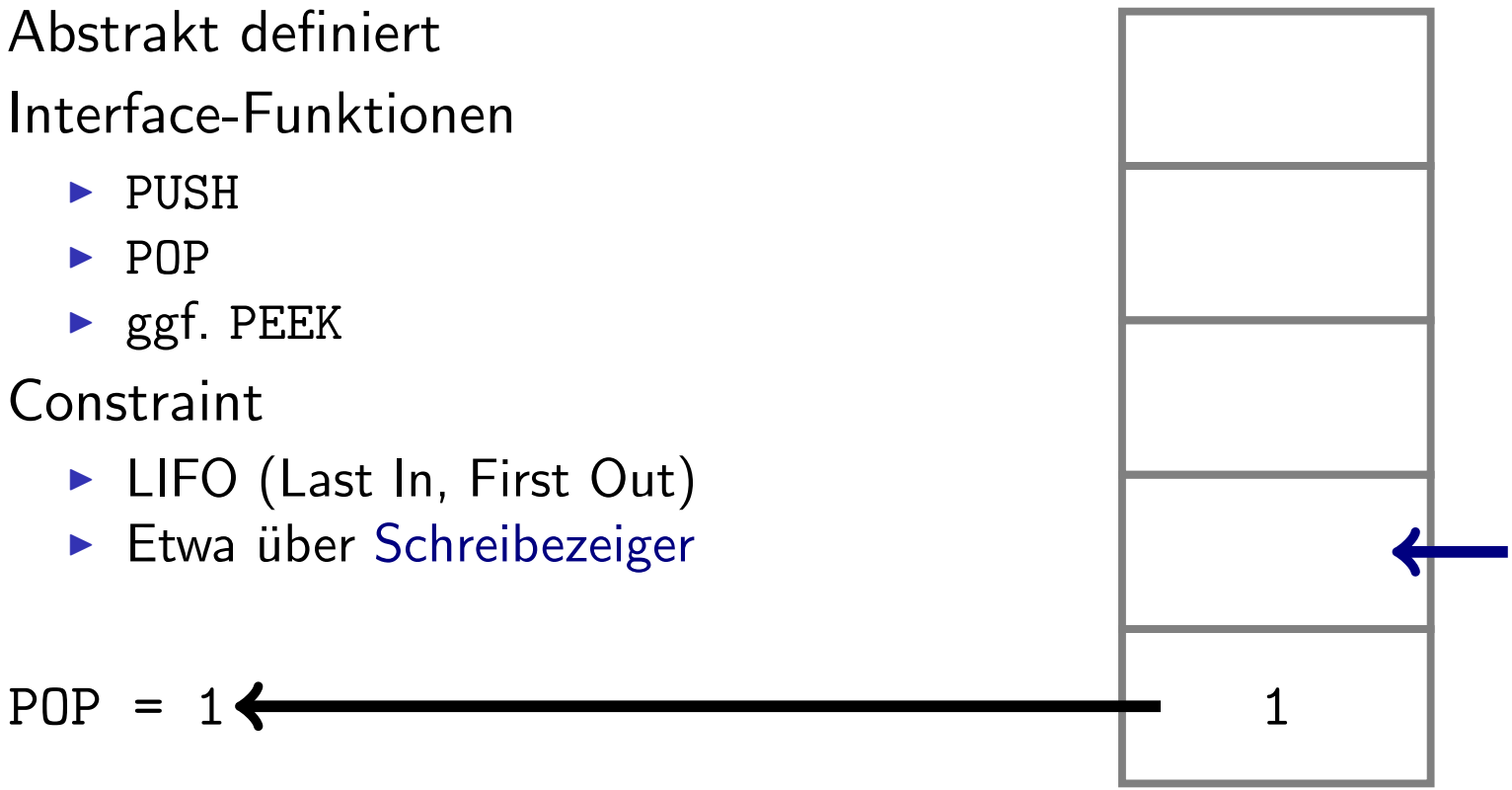
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



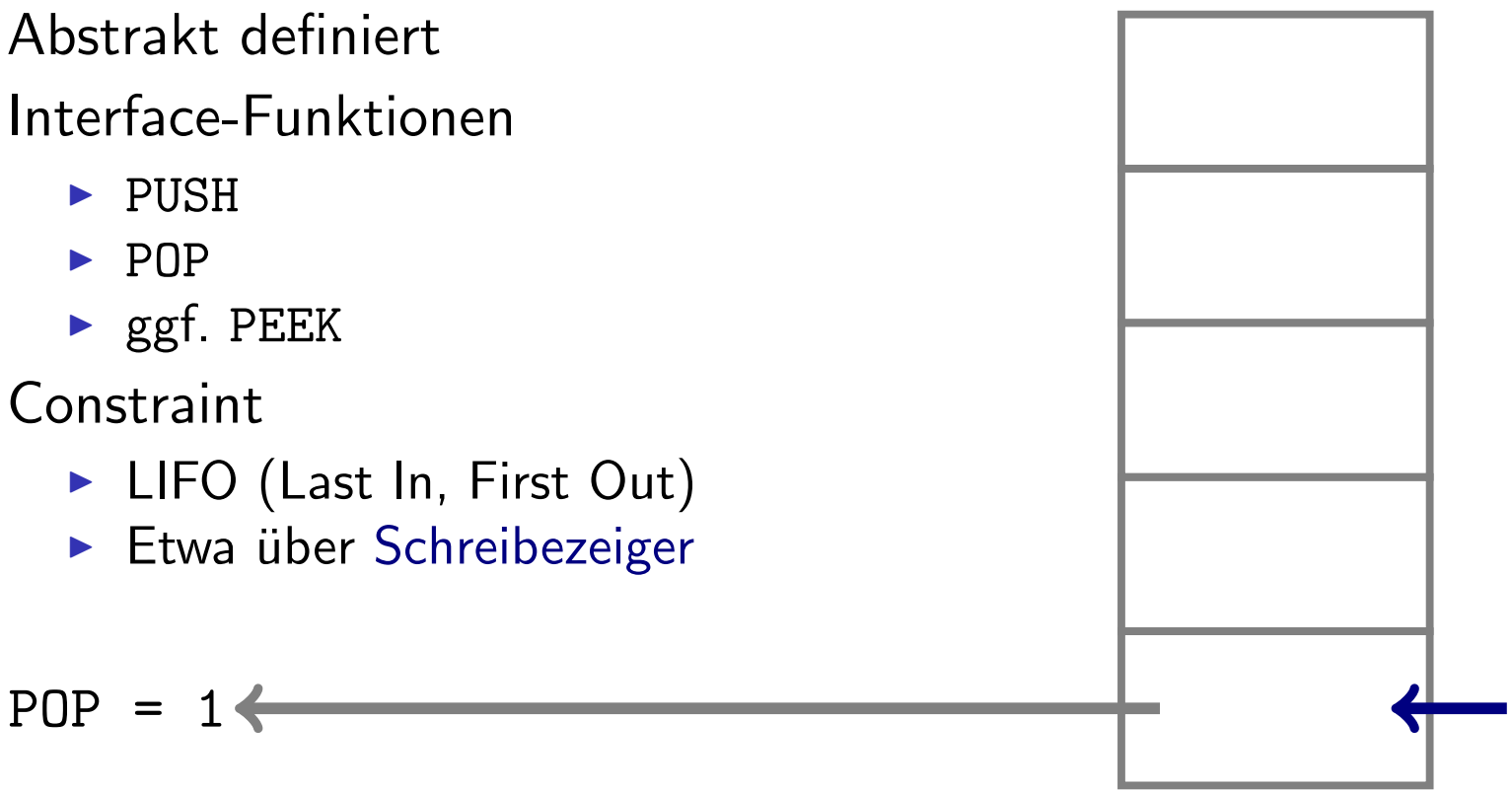
Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



Stack

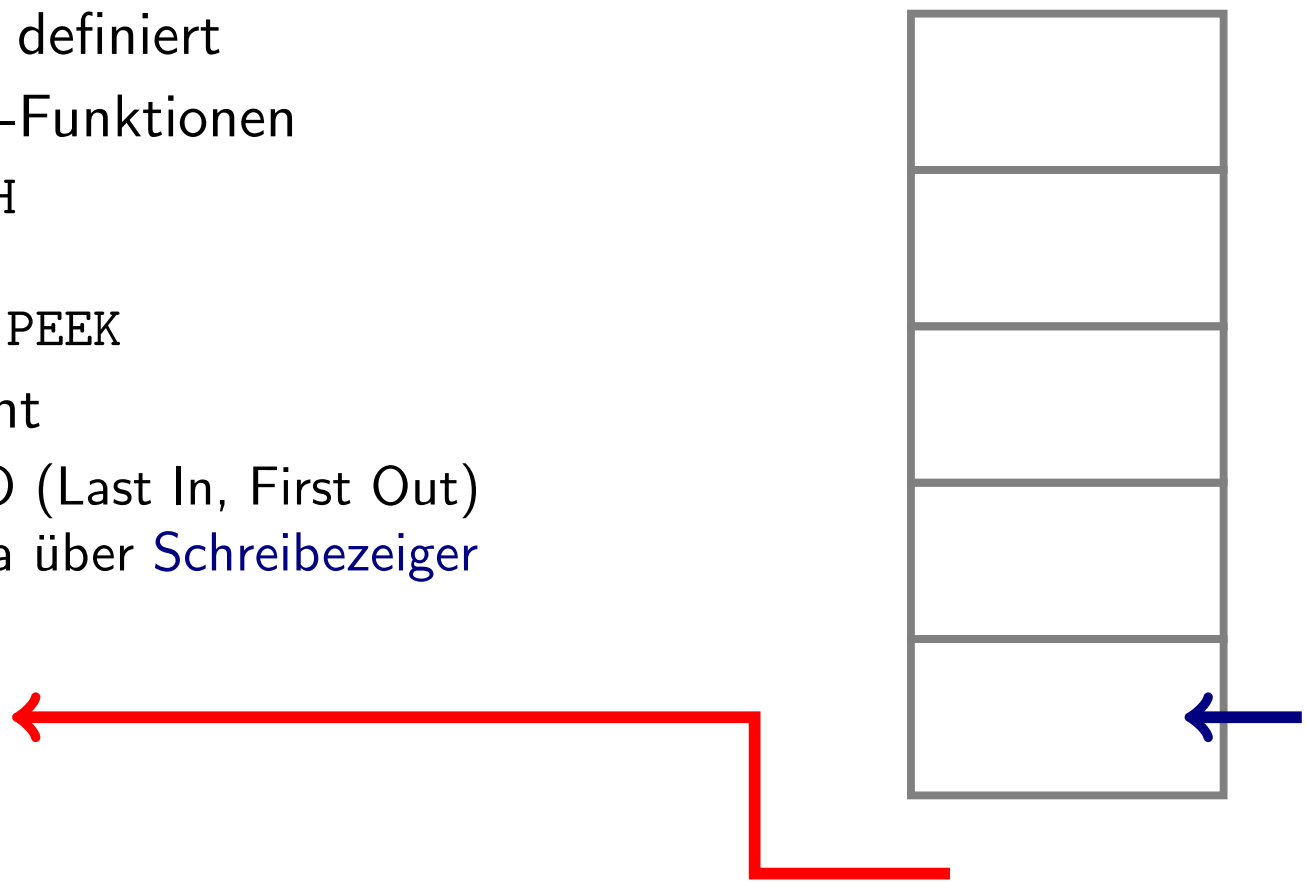
- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger



Stack

- Abstrakt definiert
- Interface-Funktionen
  - `PUSH`
  - `POP`
  - ggf. `PEEK`
- Constraint
  - LIFO (Last In, First Out)
  - Etwa über Schreibezeiger

`POP = ?`




Code

Wie wird das Programm eigentlich ausgeführt?

Ein Rechner besteht konzeptuell aus 4 Hauptkomponenten:

- Das *Steuerwerk* (*Control Unit* oder *Sequencer*) steuert den eigentlichen Ablauf aufgrund im Speicher liegender Befehle.
- Mathematische Operationen sind im *Rechenwerk* (*Execution Unit* oder *Arithmetic and Logic Unit – ALU*) ausgelagert.
- Programm und Daten liegen im *Speicherwerk* (*Memory*) mit ein-dimensionaler Adressierung.
- Die Schnittstelle zur Außenwelt ist das *Ein-/Ausgabewerk* (*Input/Output Unit – I/O Unit*).

Die ersten beiden Werke zusammen bilden den *Prozessor* (*Central Processing Unit – CPU*).

Maschinencode

- Der Compiler erzeugt Instruktionen für das Steuerwerk.
- Eine Instruktion besteht aus dem *Op Code* (oft geschrieben als *Mnemonic*) und Parametern.
- Beispiel: Aus

```
// Addiere 5 zum Wert, der bei "zahl" im Speicher
// steht, und speichere ihn an die Adresse von "wert"
int wert = zahl + 5;
```

  wird in Pseudo-Assembler etwa zu (aber dann binär codiert!):

```
ADD @zahl, 5, @wert
```

- Das Steuerwerk arbeitet diese Instruktionsliste ab, der *Befehlszähler* (*Program Counter – PC*) zeigt auf den aktuellen Rechenschritt.
- Es gibt etwa Instruktionen für Berechnungen, Verzweigungen, Speicheroperationen und Sprünge.

Speicherorganisation

Normal gibt es einen gemeinsamen Speicher für Befehle und Daten (*Von-Neumann-Architektur*), der aber in Teilen organisiert ist:

- Im *Prozess-Adressraum* liegen die kompilierten Befehle.
- *Konstante Daten* werden separat hinterlegt.
- Für lokale Variablen und Steuerinformationen dient der *(Call) Stack* („Kellerprinzip“), erfunden u. a. von F. L. Bauer, TUM!
- Weiterer Speicher steht auf dem *Heap* (Halde) zur freien Verfügung, der dort manuell verwaltet werden muss.

Motivation

- Funktionen können mit verschiedenen Parametern...
- ...von verschiedenen Stellen im Programm...
- ...und unter Umständen sogar rekursiv aufgerufen werden.

- Variablen müssen mit wenig Aufwand kollisionsfrei angelegt werden können.
- Parameter und Rückgabewerte müssen korrekt übermittelt werden.
- Das Programm muss nach der Funktion an der richtigen Stelle weitermachen.

Beispiel: C/C++ Code

```
int summe(int a, int b)
{
    int c = a + b;
    return c;
}

int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

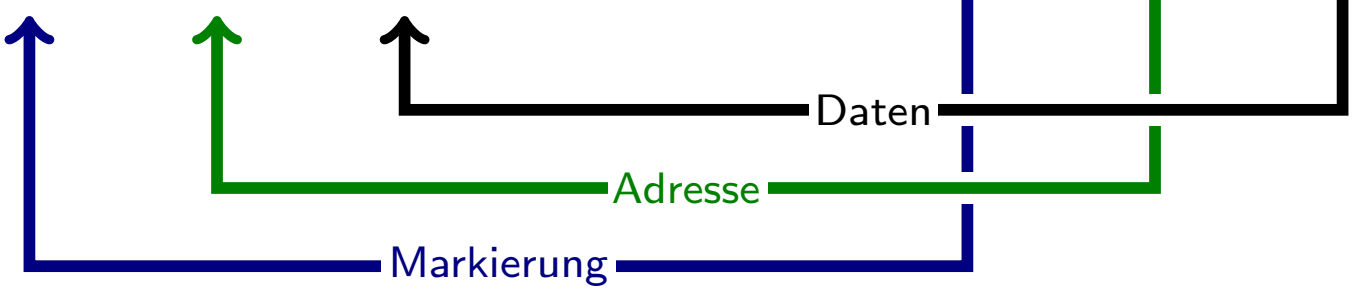
Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0000                          sp =    1000

main:   0000  PUSH @data              stack:  1000  73942
        0001  ADD sp,1,sp                     1001  23283
        0002  ADD sp,1,sp                     1002  68203
        0003  PUSH @(sp-3)                    1003  93752
        0004  PUSH 5                          1004  97492
        0005  CALL summe                     1005  18352
        0006  SUBTRACT sp,2,sp               1006  59235
        0007  POP @(sp-2)                    1007  39582
        ....  ...                            1008  82734
        ....  HALT                           ....  .....

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```



Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0000                                sp =    1000

main:   0000  PUSH @data                    stack:  1000  73942
        0001  ADD sp,1,sp                           1001  23283
        0002  ADD sp,1,sp                           1002  68203
        0003  PUSH @(sp-3)                         1003  93752
        0004  PUSH 5                               1004  97492
        0005  CALL summe                          1005  18352
        0006  SUBTRACT sp,2,sp                    1006  59235
        0007  POP @(sp-2)                         1007  39582
        ....  ...                                 1008  82734
        ....  HALT                                ....  .....

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

Programm-Adressraum: Kompilierte Befehle

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0000                          sp =    1000

main:   0000  PUSH @data              stack:  1000  73942
        0001  ADD sp,1,sp                     1001  23283
        0002  ADD sp,1,sp                     1002  68203
        0003  PUSH @(sp-3)                    1003  93752
        0004  PUSH 5                          1004  97492
        0005  CALL summe                     1005  18352
        0006  SUBTRACT sp,2,sp               1006  59235
        0007  POP @(sp-2)                    1007  39582
        ....  ...                            1008  82734
        ....  HALT                           ....  .....

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

Konstante Daten: Strings, Array-Initialisierung, ...

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =     0000                                  sp =     1000

main:    0000   PUSH @data                     stack:   1000   73942
         0001   ADD sp,1,sp                             1001   23283
         0002   ADD sp,1,sp                             1002   68203
         0003   PUSH @(sp-3)                            1003   93752
         0004   PUSH 5                                  1004   97492
         0005   CALL summe                              1005   18352
         0006   SUBTRACT sp,2,sp                        1006   59235
         0007   POP @(sp-2)                             1007   39582
         ....   ...                                     1008   82734
         ....   HALT                                    ....   .....

summe:   0100   ADD sp,1,sp
         0101   ADD @(sp-4),@(sp-3),@(sp-1)
         0102   STORE @(sp-1),@(sp-5)
         0103   SUBTRACT sp,1,sp
         0104   RETURN

data:    0200   12340
```

Call-Stack: Lokale Variablen, Funktionsparameter, Rückgaben, Kontroll-Informationen, anfänglich „Daten-Müll“ von vorherigen Anwendungen

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0000                          sp =    1000

main:   0000  PUSH @data              stack:  1000  73942
        0001  ADD sp,1,sp                     1001  23283
        0002  ADD sp,1,sp                     1002  68203
        0003  PUSH @(sp-3)                    1003  93752
        0004  PUSH 5                         1004  97492
        0005  CALL summe                     1005  18352
        0006  SUBTRACT sp,2,sp               1006  59235
        0007  POP @(sp-2)                    1007  39582
        ....  ...                           1008  82734
        ....  HALT                          ....  .....

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

Program Counter (Register): Position des auszuführenden Befehls, normalerweise kleine Inkrements, außer bei Sprüngen

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0000                                  sp =    1000

main:   0000  PUSH @data                stack:  1000  73942
        0001  ADD sp,1,sp                       1001  23283
        0002  ADD sp,1,sp                       1002  68203
        0003  PUSH @(sp-3)                      1003  93752
        0004  PUSH 5                            1004  97492
        0005  CALL summe                       1005  18352
        0006  SUBTRACT sp,2,sp                 1006  59235
        0007  POP @(sp-2)                      1007  39582
        ....  ...                              1008  82734
        ....  HALT                             ....  .....

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

Stack Pointer (Register): Schreibe-Position

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =     0000

main:    0000  PUSH @data
         0001   ADD sp,1,sp
         0002   ADD sp,1,sp
         0003   PUSH @(sp-3)
         0004   PUSH 5
         0005   CALL summe
         0006   SUBTRACT sp,2,sp
         0007   POP @(sp-2)
         ....   ...
         ....   HALT

summe:   0100   ADD sp,1,sp
         0101   ADD @(sp-4),@(sp-3),@(sp-1)
         0102   STORE @(sp-1),@(sp-5)
         0103   SUBTRACT sp,1,sp
         0104   RETURN

data:    0200   12340
```

```
sp =     1000

stack:   1000  73942
         1001   23283
         1002   68203
         1003   93752
         1004   97492
         1005   18352
         1006   59235
         1007   39582
         1008   82734
         ....   .....
```

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Legt die lokale Variable `wert` im Stack an, wobei der Wert aus dem Pool konstanter Werte (bei Marke `data:`) übertragen wird.

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =     0001

main:    0000   PUSH @data
         0001   ADD sp,1,sp
         0002   ADD sp,1,sp
         0003   PUSH @(sp-3)
         0004   PUSH 5
         0005   CALL summe
         0006   SUBTRACT sp,2,sp
         0007   POP @(sp-2)
         ....   ...
         ....   HALT

summe:   0100   ADD sp,1,sp
         0101   ADD @(sp-4),@(sp-3),@(sp-1)
         0102   STORE @(sp-1),@(sp-5)
         0103   SUBTRACT sp,1,sp
         0104   RETURN

data:    0200   12340
```

```
sp =     1001

stack:   1000   12340   # wert = 12340
         1001   23283
         1002   68203
         1003   93752
         1004   97492
         1005   18352
         1006   59235
         1007   39582
         1008   82734
         ....   .....
```

Reserviert Platz für die lokale Variable erg im Stack. Ein Wert wurde noch nicht definiert, und die Übertragung einer 0 wird eingespart.

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0002

main:   0000  PUSH @data
        0001  ADD sp,1,sp
        0002  ADD sp,1,sp
        0003  PUSH @(sp-3)
        0004  PUSH 5
        0005  CALL summe
        0006  SUBTRACT sp,2,sp
        0007  POP @(sp-2)
        ....  ...
        ....  HALT

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

```
sp =    1002

stack:  1000  12340   # wert = 12340
        1001  23283   # erg = ?
        1002  68203
        1003  93752
        1004  97492
        1005  18352
        1006  59235
        1007  39582
        1008  82734
        ....  .....
```

Reserviert Platz für den Rückgabewert der bald aufzurufenden Funktion summe, ohne zu initialisieren.

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0003

main:   0000  PUSH @data
        0001  ADD sp,1,sp
        0002  ADD sp,1,sp
        0003  PUSH @(sp-3)
        0004  PUSH 5
        0005  CALL summe
        0006  SUBTRACT sp,2,sp
        0007  POP @(sp-2)
        ....  ...
        ....  HALT

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

```
sp =    1003

stack:  1000  12340   # wert = 12340
        1001  23283   # erg = ?
        1002  68203   # rückgabe = ?
        1003  93752
        1004  97492
        1005  18352
        1006  59235
        1007  39582
        1008  82734
        ....  .....
```

Legt den ersten Parameter a auf den Stack, wobei aus dem Speicherplatz der Variable wert gelesen wird.

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0004                          sp =    1004

main:   0000  PUSH @data              stack:  1000  12340  # wert = 12340
        0001  ADD sp,1,sp                     1001  23283  # erg = ?
        0002  ADD sp,1,sp                     1002  68203  # rückgabe = ?
        0003  PUSH @(sp-3)                    1003  12340  # a = wert
        0004  PUSH 5                          1004  97492
        0005  CALL summe                      1005  18352
        0006  SUBTRACT sp,2,sp                1006  59235
        0007  POP @(sp-2)                     1007  39582
        ....  ...                            1008  82734
        ....  HALT                           ....  .....

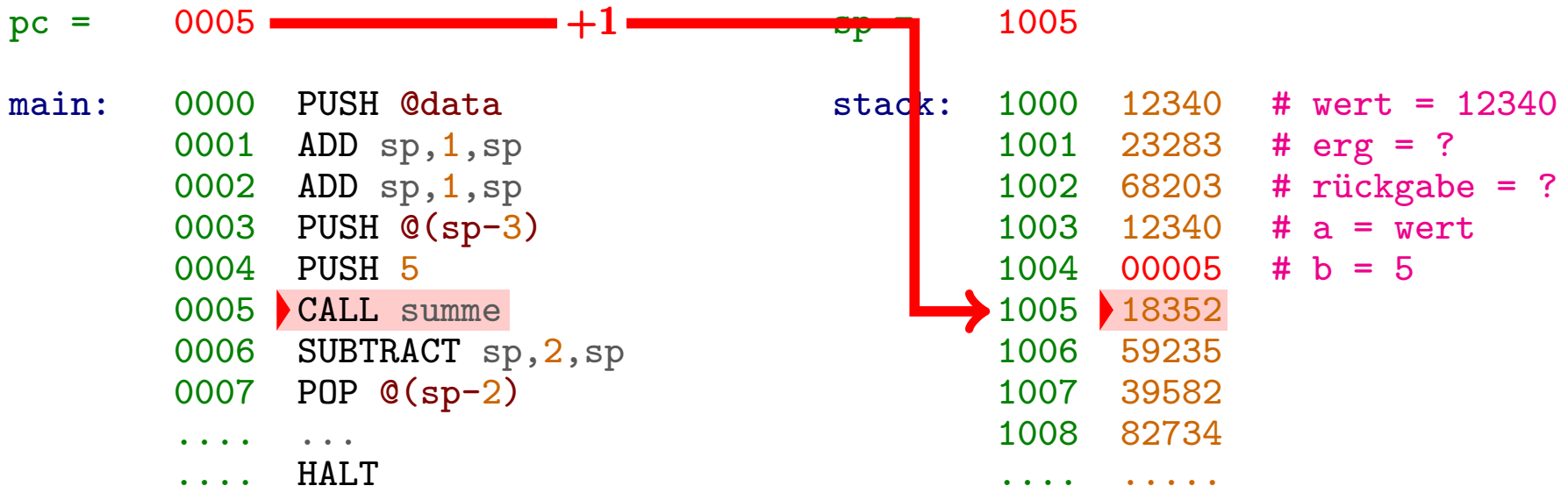
summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

Legt den zweiten Parameter b auf den Stack, wobei der kurze konstante Wert 5 diesmal (realistischerweise) direkt im Programmcode liegt.

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode



Sichert den **nächsten** Wert des Befehlszählers auf den Stack als Rücksprungadresse und verzweigt dann in die Funktion durch Änderung von pc auf summe:.

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =     0100

main:    0000   PUSH @data
         0001   ADD sp,1,sp
         0002   ADD sp,1,sp
         0003   PUSH @(sp-3)
         0004   PUSH 5
         0005   CALL summe
         0006   SUBTRACT sp,2,sp
         0007   POP @(sp-2)
         ....   ...
         ....   HALT

summe:   0100   ADD sp,1,sp
         0101   ADD @(sp-4),@(sp-3),@(sp-1)
         0102   STORE @(sp-1),@(sp-5)
         0103   SUBTRACT sp,1,sp
         0104   RETURN

data:    0200   12340
```

```
sp =     1006

stack:   1000   12340   # wert = 12340
         1001   23283   # erg = ?
         1002   68203   # rückgabe = ?
         1003   12340   # a = wert
         1004   00005   # b = 5
         1005   00006   # rücksprung-adr.
         1006   59235
         1007   39582
         1008   82734
         ....   .....
```

```
int summe(int a, int b)
{
    int c = a + b;
    return c;
}
```

Reserviert Platz für die lokale Variable c im Stack, ohne zu initialisieren.

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0101

main:   0000  PUSH @data
        0001  ADD sp,1,sp
        0002  ADD sp,1,sp
        0003  PUSH @(sp-3)
        0004  PUSH 5
        0005  CALL summe
        0006  SUBTRACT sp,2,sp
        0007  POP @(sp-2)
        ....  ...
        ....  HALT

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

```
sp =    1007

stack:  1000  12340   # wert = 12340
        1001  23283   # erg = ?
        1002  68203   # rückgabe = ?
        1003  12340   # a = wert
        1004  00005   # b = 5
        1005  00006   # rücksprung-adr.
        1006  59235   # c = ?
        1007  39582
        1008  82734
        ....  .....
```

```
int summe(int a, int b)
{
    int c = a + b;
    return c;
}
```

Berechnet die Summe aus den beiden Parametern a und b, und schreibt das Ergebnis in die Zelle der Variable c.

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0102

main:   0000  PUSH @data
        0001  ADD sp,1,sp
        0002  ADD sp,1,sp
        0003  PUSH @(sp-3)
        0004  PUSH 5
        0005  CALL summe
        0006  SUBTRACT sp,2,sp
        0007  POP @(sp-2)
        ....  ...
        ....  HALT

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

```
sp =    1007

stack:  1000  12340   # wert = 12340
        1001  23283   # erg = ?
        1002  68203   # rückgabe = ?
        1003  12340   # a = wert
        1004  00005   # b = 5
        1005  00006   # rücksprung-adr.
        1006  12345   # c = 12345
        1007  39582
        1008  82734
        ....  .....
```

```
int summe(int a, int b)
{
    int c = a + b;
    return c;
}
```

Der Wert von c soll als Rückgabe der Funktion verwendet werden. Er wird daher in die zuvor reservierte Rückgabezelle kopiert.

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0103                          sp =    1007

main:   0000  PUSH @data              stack:  1000  12340  # wert = 12340
        0001  ADD sp,1,sp                     1001  23283  # erg = ?
        0002  ADD sp,1,sp                     1002  12345  # rückgabe = 12345
        0003  PUSH @(sp-3)                    1003  12340  # a = wert
        0004  PUSH 5                          1004  00005  # b = 5
        0005  CALL summe                      1005  00006  # rücksprung-adr.
        0006  SUBTRACT sp,2,sp                1006  12345  # c = 12345
        0007  POP @(sp-2)                    1007  39582
        ....  ...                           1008  82734
        ....  HALT                          ....  .....

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

```
int summe(int a, int b)
{
    int c = a + b;
    return c;
}
```

Die lokale Variable c ist nun nicht mehr notwendig.
Die Löschung erfolgt durch Dekrement des Schreibezeigers, ein neuer Wert wird nicht geschrieben.

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =     0104                    sp =     1006

main:    0000   PUSH @data       stack:   1000   12340   # wert = 12340
         0001   ADD sp,1,sp              1001   23283   # erg = ?
         0002   ADD sp,1,sp              1002   12345   # rückgabe = 12345
         0003   PUSH @(sp-3)             1003   12340   # a = wert
         0004   PUSH 5                   1004   00005   # b = 5
         0005   CALL summe               1005   00006   # rücksprung-adr.
         0006   SUBTRACT sp,2,sp         1006   12345
         0007   POP @(sp-2)              1007   39582
         ....   ...                      1008   82734
         ....   HALT                     ....   .....

summe:   0100   ADD sp,1,sp
         0101   ADD @(sp-4),@(sp-3),@(sp-1)
         0102   STORE @(sp-1),@(sp-5)
         0103   SUBTRACT sp,1,sp
         0104   RETURN

data:    0200   12340
```

```
int summe(int a, int b)
{
    int c = a + b;
    return c;
}
```

Nach diesen Aufräumarbeiten kann die Funktion verlassen werden. Dazu wird der Befehlszähler auf den zuvor gespeicherten Rücksprungzeiger gesetzt.

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0006

main:   0000  PUSH @data
        0001  ADD sp,1,sp
        0002  ADD sp,1,sp
        0003  PUSH @(sp-3)
        0004  PUSH 5
        0005  CALL summe
        0006  SUBTRACT sp,2,sp
        0007  POP @(sp-2)
        ....  ...
        ....  HALT

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

```
sp =    1005

stack:  1000  12340  # wert = 12340
        1001  23283  # erg = ?
        1002  12345  # rückgabe = 12345
        1003  12340  # a = wert
        1004  00005  # b = 5
        1005  00006
        1006  12345
        1007  39582
        1008  82734
        ....  .....
```

Die beiden Parameter der Funktion sind nun nicht mehr notwendig und werden – analog zu c – durch Dekrement des Stack Pointers gelöscht.

```c
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =    0007

main:   0000  PUSH @data
        0001  ADD sp,1,sp
        0002  ADD sp,1,sp
        0003  PUSH @(sp-3)
        0004  PUSH 5
        0005  CALL summe
        0006  SUBTRACT sp,2,sp
        0007  POP @(sp-2)
        ....  ...
        ....  HALT

summe:  0100  ADD sp,1,sp
        0101  ADD @(sp-4),@(sp-3),@(sp-1)
        0102  STORE @(sp-1),@(sp-5)
        0103  SUBTRACT sp,1,sp
        0104  RETURN

data:   0200  12340
```

```
sp =    1003

stack:  1000  12340   # wert = 12340
        1001  12345   # erg = ?
        1002  12345   # rückgabe = 12345
        1003  12340
        1004  00005
        1005  00006
        1006  12345
        1007  39582
        1008  82734
        ....  .....
```

Der Rückgabewert muss nun nach erg kopiert werden, bevor auch dieser gelöscht werden kann. Dies geschieht durch eine einzige POP-Operation.

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Naïve Übersetzung in Pseudo-Maschinencode

```
pc =     ....

main:    0000   PUSH @data
         0001   ADD sp,1,sp
         0002   ADD sp,1,sp
         0003   PUSH @(sp-3)
         0004   PUSH 5
         0005   CALL summe
         0006   SUBTRACT sp,2,sp
         0007   POP @(sp-2)
         ....   ...
         ....   HALT

summe:   0100   ADD sp,1,sp
         0101   ADD @(sp-4),@(sp-3),@(sp-1)
         0102   STORE @(sp-1),@(sp-5)
         0103   SUBTRACT sp,1,sp
         0104   RETURN

data:    0200   12340
```

```
sp =     1002

stack:   1000   12340   # wert = 12340
         1001   12345   # erg = 12345
         1002   12345
         1003   12340
         1004   00005
         1005   00006
         1006   12345
         1007   39582
         1008   82734
         ....   .....
```

Nun ist der Funktionsaufruf abgeschlossen, wobei die Daten als neuer „Müll“ im Speicher verbleiben. Die beiden Variablen sind gesetzt wie erwartet.

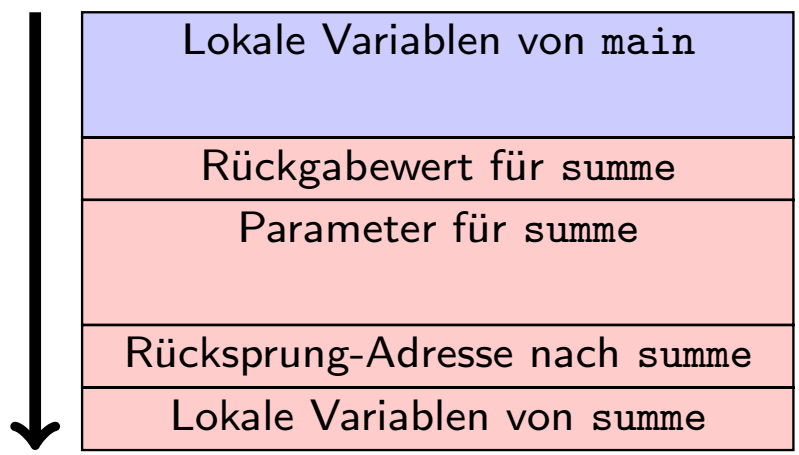
```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

Beispiel: Stack-Organisation

```
int main(...)
{
    int wert = 12340;
    int erg = summe(wert, 5);
    // ...
    return 0;
}
```

```
int summe(int a, int b)
{
    int c = a + b;
    return c;
}
```

```
stack:  1000  .....  # wert
        1001  .....  # erg
        1002  .....  # rückgabe-wert
        1003  .....  # a
        1004  .....  # b
        1005  .....  # rücksprung-adr.
        1006  .....  # c
        1007  .....
        1008  .....
        ....  .....
```



Rekursion ist damit (im Rahmen der Kapazität des Call Stacks) kein Problem! Es werden einfach entsprechende weitere Einträge erzeugt.

Caveat

Wieder einmal ist die Betrachtung vereinfacht:

- Unser Prozessor hat keine Universalregister.
- Unsere Adressen sind eher Einheitenzähler und betrachten keine Datentypen, Längen von Maschinenbefehlen, etc.
- Die Anordnung auf dem echten Call Stack ist ähnlich, aber es existieren Unterschiede.
- Der Machinencode ist nicht optimiert.
  - Gute Compiler haben enorme „Intelligenz“!

Konsequenzen

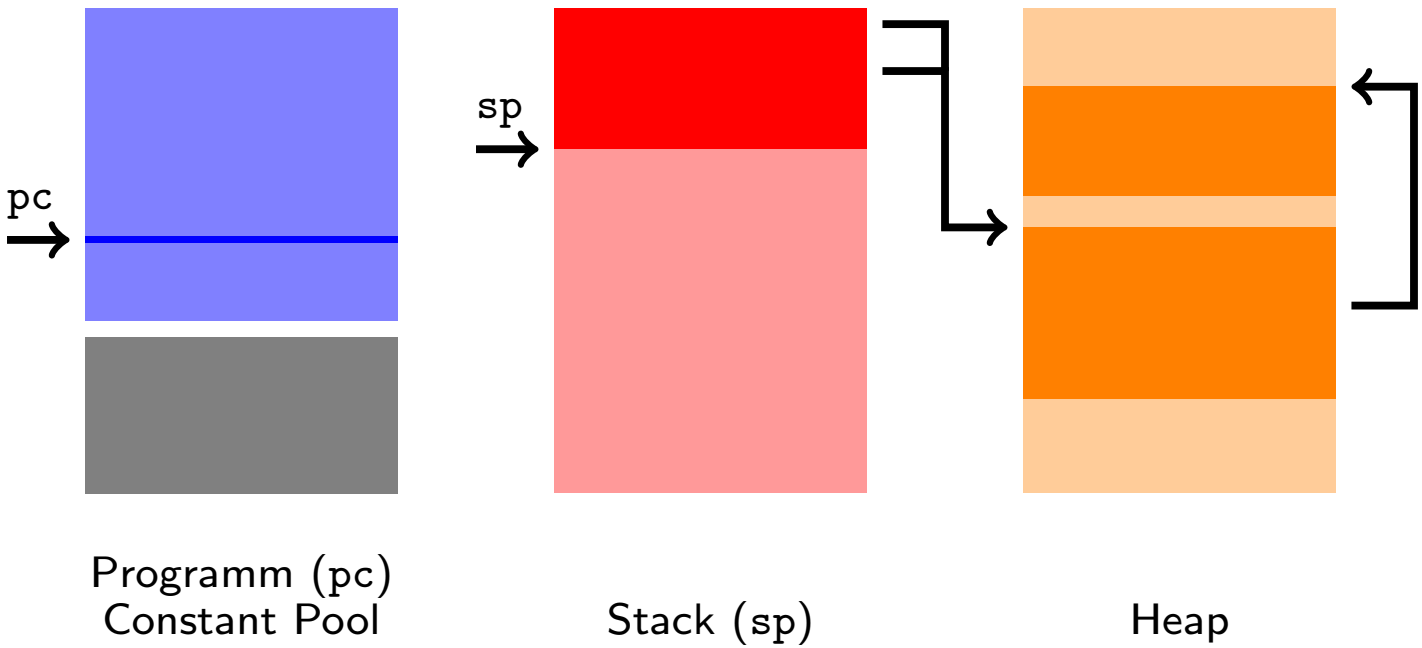
- In C (nicht C++) müssen alle lokalen Variablen schon zu Beginn der Funktion definiert werden.
- Lokale Variablen haben einen Gültigkeitsbereich (*Scope*).
- Lokale Variablen müssen immer fixe Länge haben, damit die sp-relative Indexberechnung korrekt ist.

```
// Anzahl muss zur Kompilierungszeit klar sein
int array[5];
```

- Außerhalb der Grenzen des Arrays kann sehr schnell Kontrollinformation beschädigt werden!
  - Vorlesung: `strcpy` vs. `strncpy`

- Variabel große Datenblöcke müssen anderswo liegen!

Heap



- Platz auf dem Heap muss manuell angefordert und nach Ende der Nutzung (genau einmal) freigegeben werden.
  - C : `malloc(...)` und `free(...)`
  - C++: `new`/`new[]` und `delete`/`delete[]`

Zeiger

- Verweise auf allozierten Speicher (also 32- oder 64-bit-Adressen) nennt man *Zeiger* (*Pointer*).

```
int32_t* array = new int32_t[n];

// Gebe die Adresse aus, sie verweist auf den Heap.
std::cout << array << std::endl;

// Jetzt greifen wir auf die erste Zelle zu.
// Sie liegt bei Adresse + 0 (also array + 0)
array[0] = 1;

// Die vierte liegt bei Adresse + 3 * Elementgroesse
// (also array + 3 * (4 Byte))
array[3] = 4;

delete[] array;
```

- Pointer sind lokale Variablen oder ebenfalls im Heap.
- Man kann natürlich auch die Adressen von lokalen Variablen und Funktionen herausbekommen; Arrays sind **immer** Zeiger.

Potentielle Probleme

```
// Erzeuge ein Array mit dynamischer Anzahl, und arbeite damit
int* array = new int[n];

// ...
// ...
// ...
// ...
// ...
// ...

// Der Speicherblock wird wieder freigegeben
delete[] array;
```

Potentielle Probleme

```
// Erzeuge ein Array mit dynamischer Anzahl, und arbeite damit
int* array = new int[n];

if (abbruchbedingung)
{
    // Hier wird die Funktion verlassen, ohne freizugeben:
    // SPEICHERLECK!
    return;
}

// Der Speicherblock wird wieder freigegeben
delete[] array;
```

Potentielle Probleme

- Früher konnte man mit Speicherlecks das ganze Betriebssystem „abwürgen“.
- Heute sind die Kernel schlauer und räumen „Übeltätern“ hinterher.
- Trotzdem natürlich kann das eigentliche Programm abstürzen.

- Wir werden bald eine bessere Methode zur Verwaltung sehen!