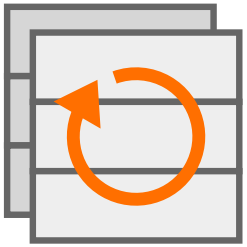


Pointer und Arrays in C

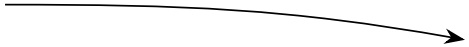
Prof. Dr.-Ing Christian Dietrich

12. Dezember 2022

Wiederholung: Werte und Variablen

Wert

- Typ → Wertemenge
- Bsp: 8-Bit Ganzzahl
  `uint8_t` → {0,..,255}
- CPU verrechnet Werte

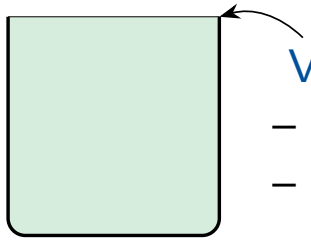


3

Wiederholung: Werte und Variablen

Wert
- Typ → Wertemenge
- Bsp: 8-Bit Ganzzahl `uint8_t` → {0,..,255}
- CPU verrechnet Werte

3



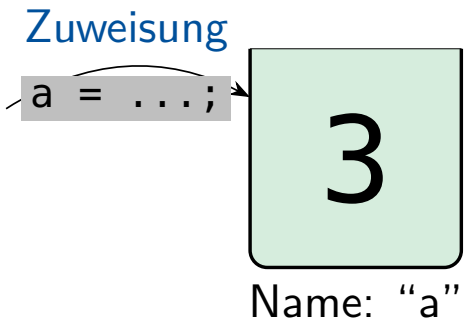
Variable
- Platz für einen Wert
- “Lebt” im Speicher
- Typ und Name im Code
- Bsp: `uint8_t a;`

Wiederholung: Werte und Variablen

Wert

- Typ → Wertemenge
- Bsp: 8-Bit Ganzzahl
  `uint8_t` → {0,..,255}
- CPU verrechnet Werte

3



Variable

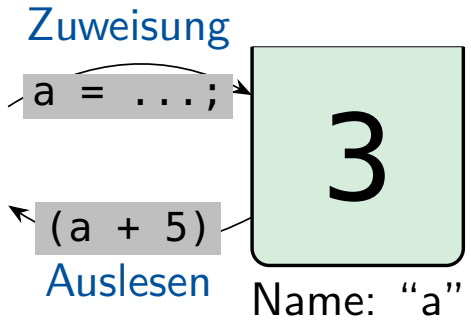
- Platz für einen Wert
- “Lebt” im Speicher
- Typ und Name im Code
- Bsp: `uint8_t a;`

Wiederholung: Werte und Variablen

Wert

- Typ → Wertemenge
- Bsp: 8-Bit Ganzzahl
  `uint8_t` → {0,..,255}
- CPU verrechnet Werte

8



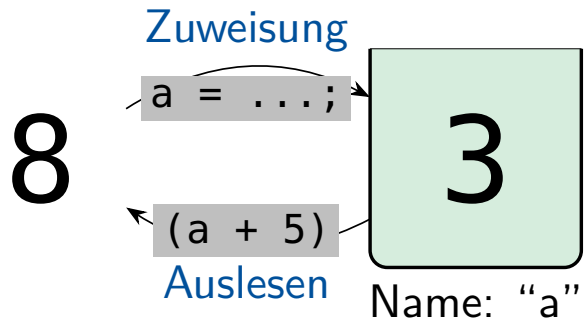
Variable

- Platz für einen Wert
- “Lebt” im Speicher
- Typ und Name im Code
- Bsp: `uint8_t a;`

Wiederholung: Werte und Variablen

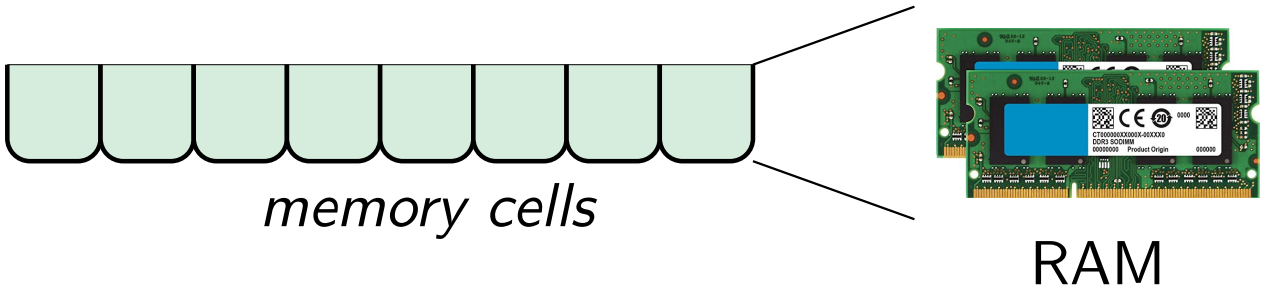
Wert

- Typ → Wertemenge
- Bsp: 8-Bit Ganzzahl
  `uint8_t` → {0,..,255}
- CPU verrechnet Werte



Variable

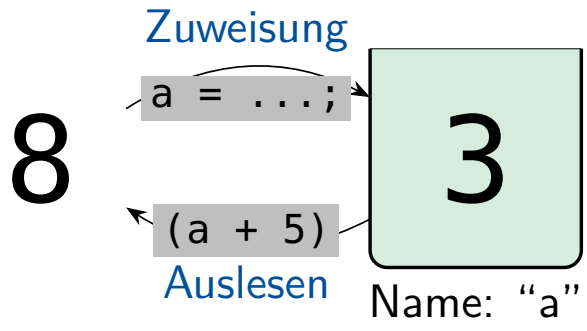
- Platz für einen Wert
- “Lebt” im Speicher
- Typ und Name im Code
- Bsp: `uint8_t a;`



Wiederholung: Werte und Variablen

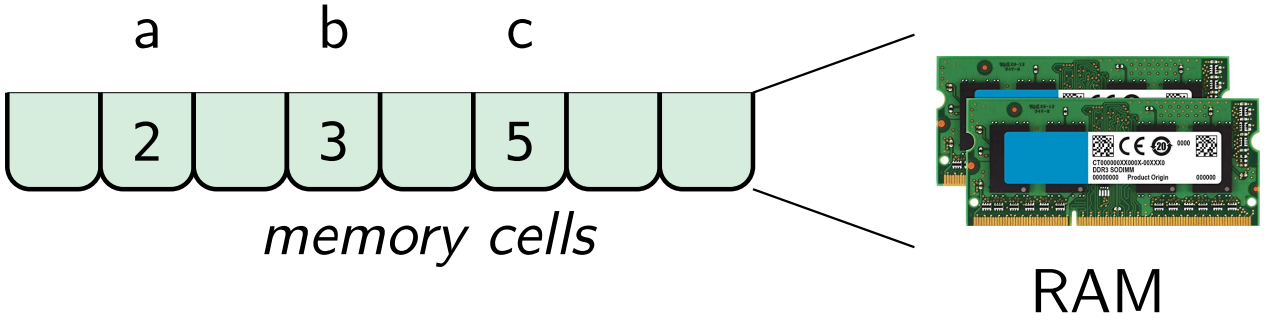
Wert

- Typ → Wertemenge
- Bsp: 8-Bit Ganzzahl
  `uint8_t` → {0,..,255}
- CPU verrechnet Werte

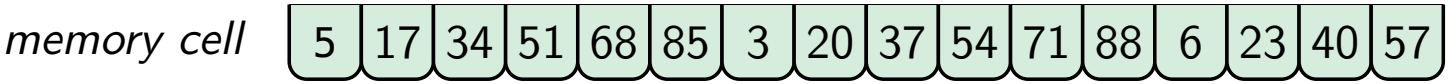


Variable

- Platz für einen Wert
- “Lebt” im Speicher
- Typ und Name im Code
- Bsp: `uint8_t a;`

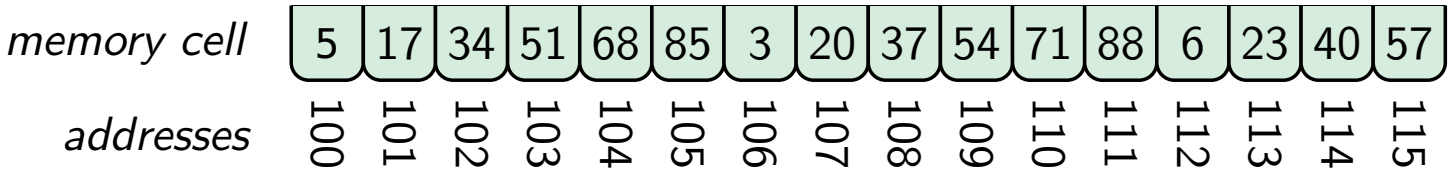


Adressen und Pointer/Zeiger



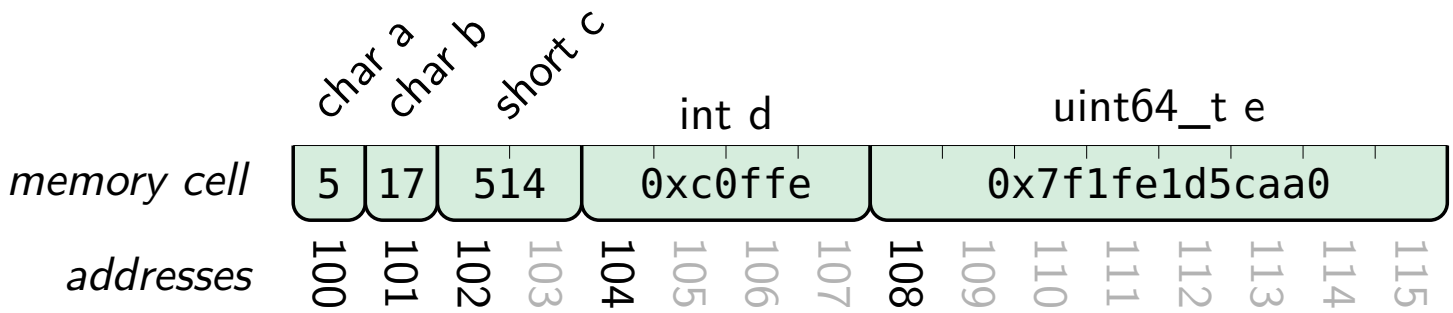
- **Problem**: 48 GiB $\approx$ 51 Mrd. `uint8_t` $\Rightarrow$ Viel zu viele Namen
  - Die Hardware verarbeitet keine symbolischen Namen

Adressen und Pointer/Zeiger



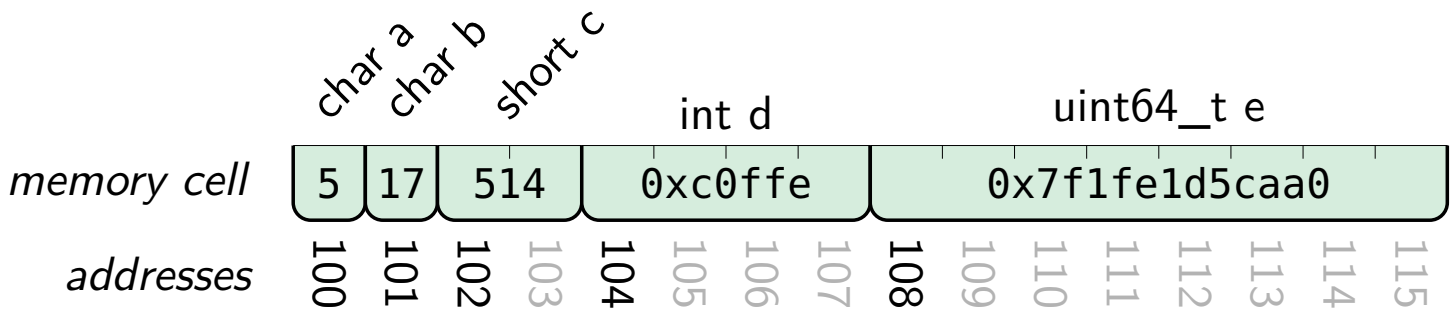
- **Problem**: 48 GiB $\approx$ 51 Mrd. `uint8_t` $\Rightarrow$ Viel zu viele Namen
  - Die Hardware verarbeitet keine symbolischen Namen
  - **Idee**: Jede Speicherzelle bekommt eine fixe Nummer $\Rightarrow$ Adresse

Adressen und Pointer/Zeiger



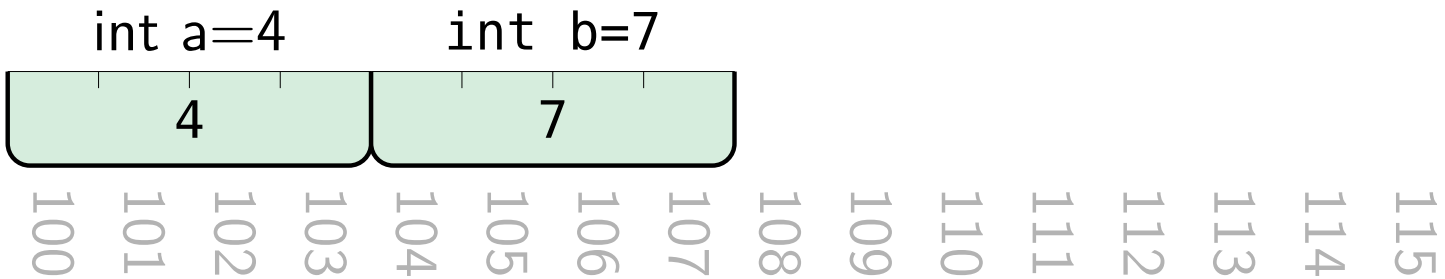
- **Problem**: 48 GiB ≈ 51 Mrd. `uint8_t` ⇒ Viel zu viele Namen
  - Die Hardware verarbeitet keine symbolischen Namen
  - **Idee**: Jede Speicherzelle bekommt eine fixe Nummer ⇒ Adresse
  - Verschieden große Typen; niedrigste Adresse identifiziert die Variable

Adressen und Pointer/Zeiger

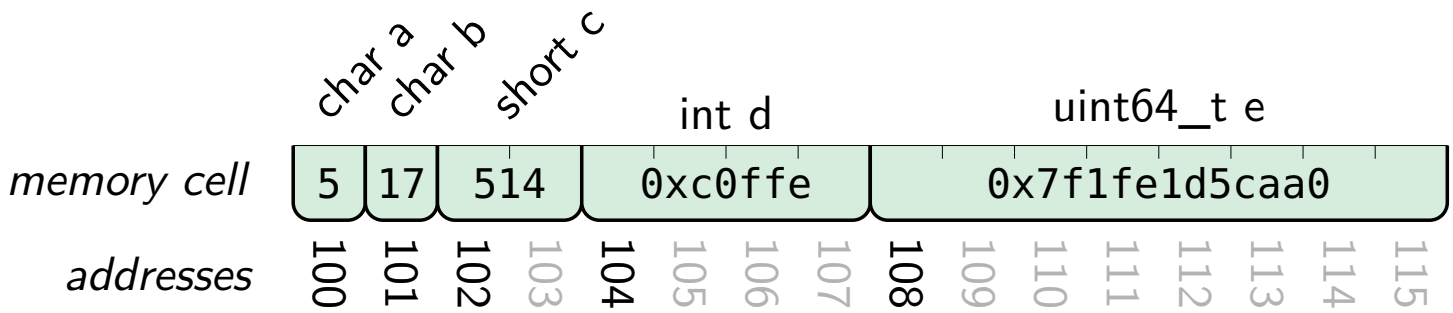


- **Problem**: 48 GiB ≈ 51 Mrd. `uint8_t` ⇒ Viel zu viele Namen
  - Die Hardware verarbeitet keine symbolischen Namen
  - **Idee**: Jede Speicherzelle bekommt eine fixe Nummer ⇒ Adresse
  - Verschieden große Typen; niedrigste Adresse identifiziert die Variable

- **Idee**: Wir speichern Adressen in Variablen. ⇒ **Pointer!**

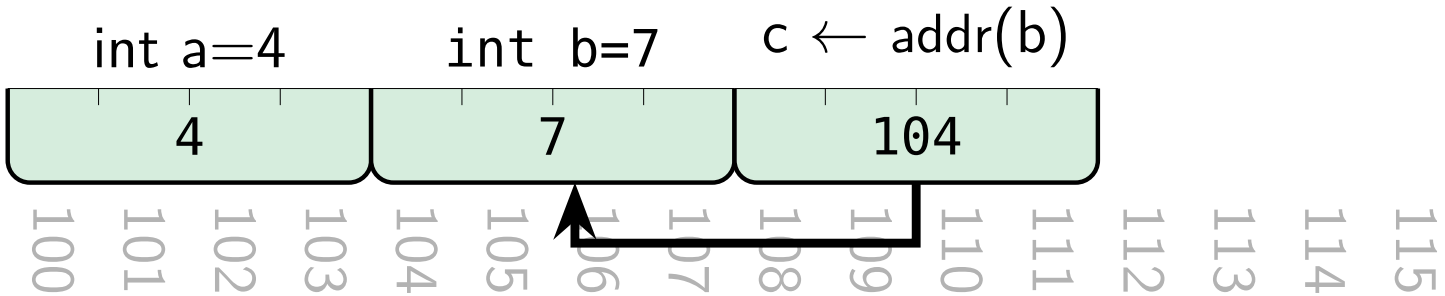


Adressen und Pointer/Zeiger

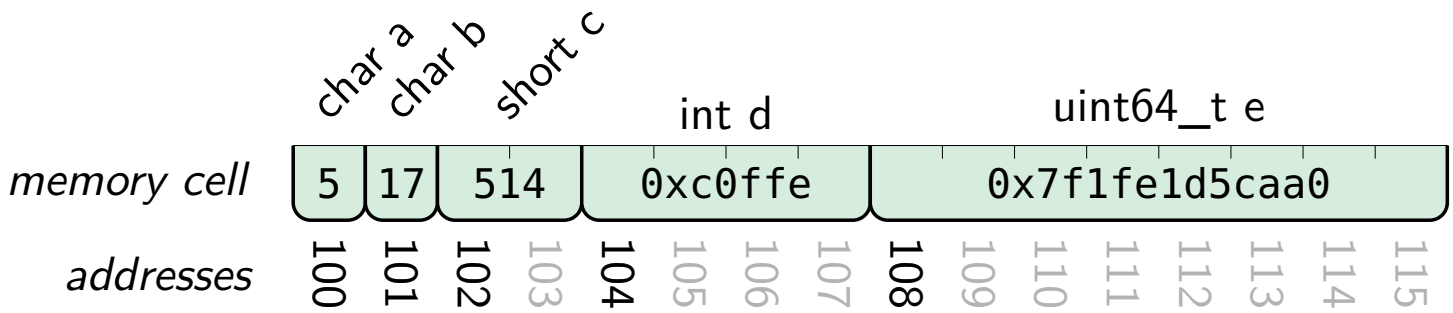


- **Problem**: 48 GiB ≈ 51 Mrd. `uint8_t` ⇒ Viel zu viele Namen
  - Die Hardware verarbeitet keine symbolischen Namen
  - **Idee**: Jede Speicherzelle bekommt eine fixe Nummer ⇒ Adresse
  - Verschieden große Typen; niedrigste Adresse identifiziert die Variable

- **Idee**: Wir speichern Adressen in Variablen. ⇒ **Pointer!**
  - "addr()": Sprache erlaubt Berechnung von Variablen-Adressen

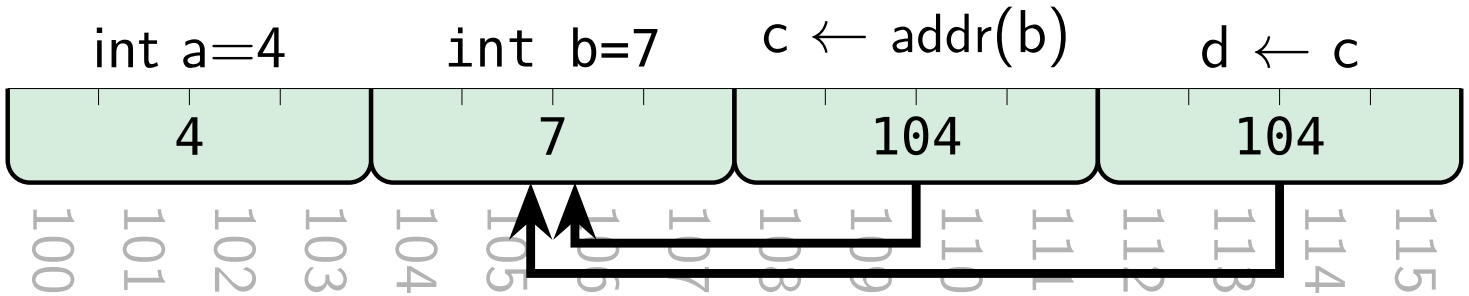


Adressen und Pointer/Zeiger



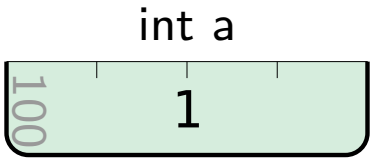
- **Problem**: 48 GiB ≈ 51 Mrd. `uint8_t` ⇒ Viel zu viele Namen
  - Die Hardware verarbeitet keine symbolischen Namen
  - **Idee**: Jede Speicherzelle bekommt eine fixe Nummer ⇒ Adresse
  - Verschieden große Typen; niedrigste Adresse identifiziert die Variable

- **Idee**: Wir speichern Adressen in Variablen. ⇒ **Pointer!**
  - "addr()": Sprache erlaubt Berechnung von Variablen-Adressen
  - Mehrere Pointer können auf dieselbe Variable zeigen.



Syntax für Pointer

```
int main(int argc, char** argv) {
  int a = 1;
  int * ptr ;

  ptr =  &a ;

   *ptr  =  *ptr  +  *ptr ;
}
```

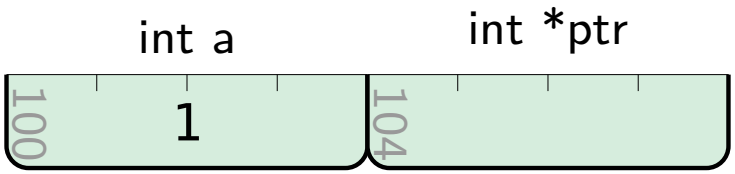


Syntax für Pointer

```
int main(int argc, char** argv) {
  int a = 1;
  int *ptr;

  ptr = &a;

  *ptr = *ptr + *ptr;
}
```

Deklaration/Definition

- `T *`: Pointer auf Typ T
- Alle Pointer sind gleich breit (4/8 B)



Syntax für Pointer

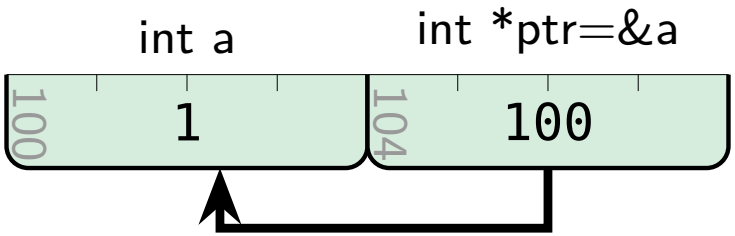
```
int main(int argc, char** argv) {
  int a = 1;
  int * ptr ;

  ptr = &a ;

   *ptr = *ptr + *ptr ;
}
```

Deklaration/Definition

- `T *` : Pointer auf Typ T
- Alle Pointer sind gleich breit (4/8 B)

address-of Operator

- `&(E)` : Adresse von Ausdruck E
- Erzeugt Pointer**wert** aus Adresse



Syntax für Pointer

```
int main(int argc, char** argv) {
  int a = 1;
  int * ptr ;

  ptr =  &a ;

   *ptr  =  *ptr  +  *ptr ;
}
```

Deklaration/Definition

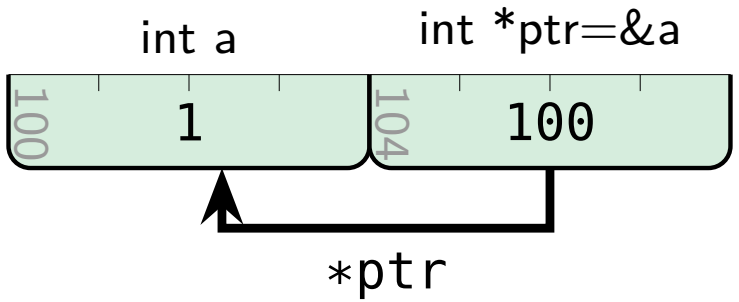
- `T *` : Pointer auf Typ T
- Alle Pointer sind gleich breit (4/8 B)

address-of Operator

- `&(E)` : Adresse von Ausdruck E
- Erzeugt Pointer**wert** aus Adresse

Dereferenz Operator

- `*(E)` : Zugriff auf Pointee (R/W)
- Läuft dem Pointer ein Level nach
- `typeof(*ptr)` → `int`



Syntax für Pointer

```
int main(int argc, char** argv) {
  int a = 1;
  int * ptr ;

  ptr =  &a ;

  *ptr  =  *ptr  +  *ptr ;
}
```

Deklaration/Definition

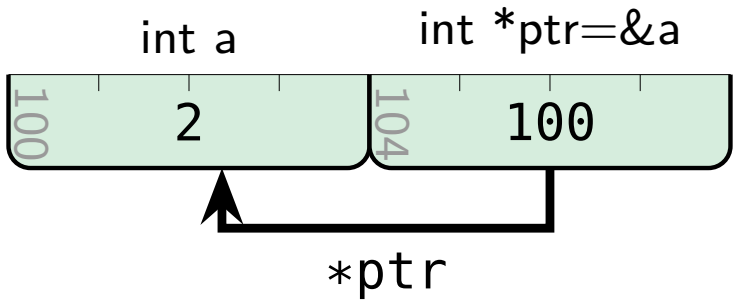
- `T *` : Pointer auf Typ T
- Alle Pointer sind gleich breit (4/8 B)

address-of Operator

- `&(E)` : Adresse von Ausdruck E
- Erzeugt Pointer**wert** aus Adresse

Dereferenz Operator

- `*(E)` : Zugriff auf Pointee (R/W)
- Läuft dem Pointer ein Level nach
- `typeof(*ptr)` → `int`



Syntax für Pointer

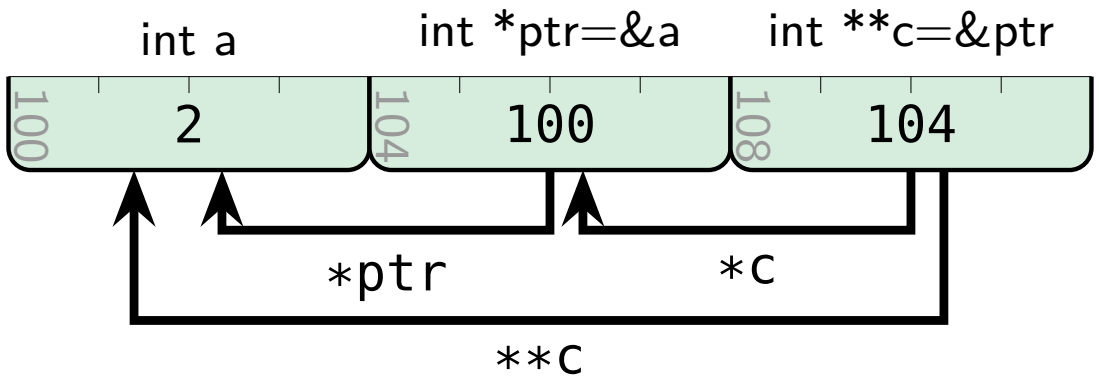
```
int main(int argc, char** argv) {
  int a = 1;
  int * ptr ;

  ptr =  &a ;

   *ptr  =  *ptr  +  *ptr ;
}
```

Deklaration/Definition
- `T *` : Pointer auf Typ T
- Alle Pointer sind gleich breit (4/8 B)

address-of Operator
- `&(E)` : Adresse von Ausdruck E
- Erzeugt Pointer**wert** aus Adresse

Dereferenz Operator
- `*(E)` : Zugriff auf Pointee (R/W)
- Läuft dem Pointer ein Level nach
- `typeof(*ptr)` → `int`

- Pointer auf Pointer-Variablen
  - `int** x` : Pointer → int-Pointer
  - `typeof(*c)` = `int*`



Syntax für Pointer

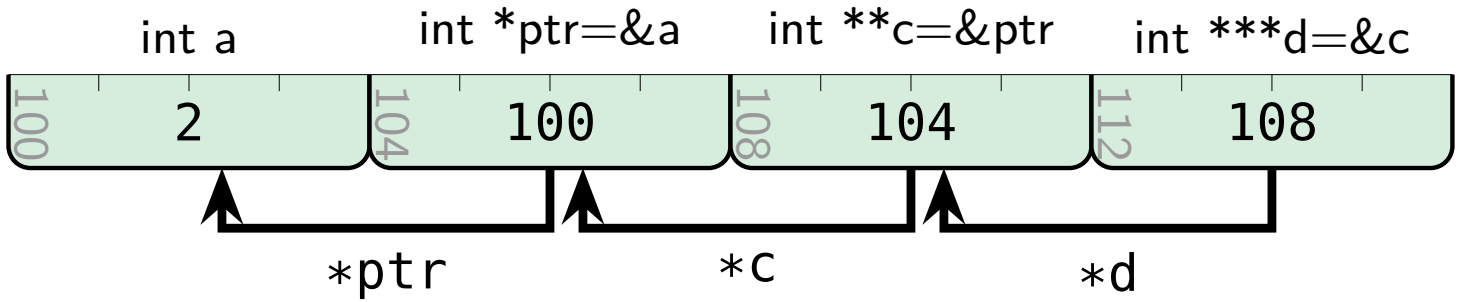
```
int main(int argc, char** argv) {
  int a = 1;
  int * ptr ;

  ptr =  &a ;

   *ptr  =  *ptr  +  *ptr ;
}
```

Deklaration/Definition
- `T *` : Pointer auf Typ T
- Alle Pointer sind gleich breit (4/8 B)

address-of Operator
- `&(E)` : Adresse von Ausdruck E
- Erzeugt Pointer**wert** aus Adresse

Dereferenz Operator
- `*(E)` : Zugriff auf Pointee (R/W)
- Läuft dem Pointer ein Level nach
- `typeof(*ptr)` → `int`

- Pointer auf Pointer-Variablen
  - `int** x` : Pointer → int-Pointer
  - `typeof(*c)` = `int*`



Syntax für Pointer

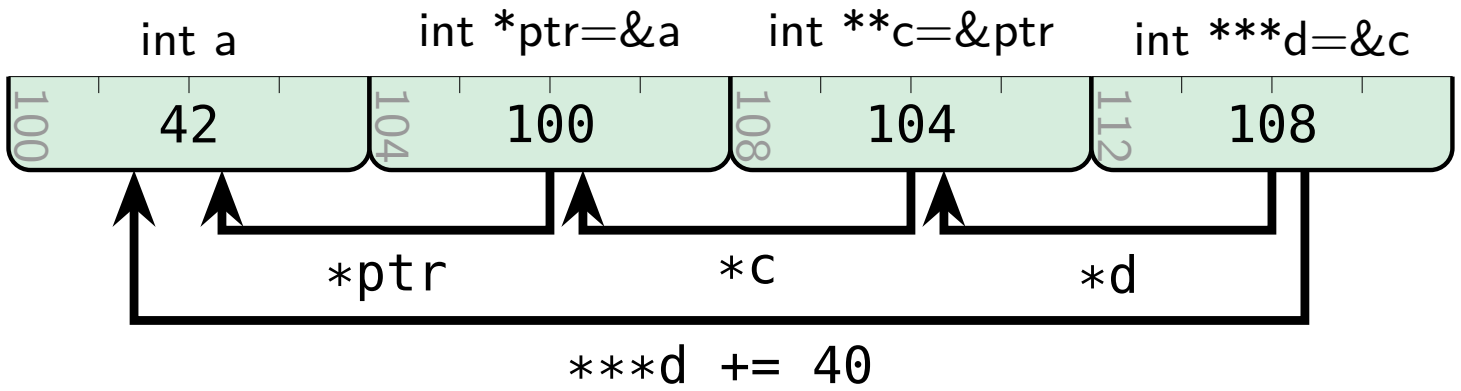
```
int main(int argc, char** argv) {
  int a = 1;
  int * ptr ;

  ptr =  &a ;

   *ptr  =  *ptr  +  *ptr ;
}
```

Deklaration/Definition
- `T *` : Pointer auf Typ T
- Alle Pointer sind gleich breit (4/8 B)

address-of Operator
- `&(E)` : Adresse von Ausdruck E
- Erzeugt Pointer**wert** aus Adresse

Dereferenz Operator
- `*(E)` : Zugriff auf Pointee (R/W)
- Läuft dem Pointer ein Level nach
- `typeof(*ptr)` → `int`

- Pointer auf Pointer-Variablen
  - `int** x` : Pointer → int-Pointer
  - `typeof(*c)` = `int*`



Rechnen mit Pointern

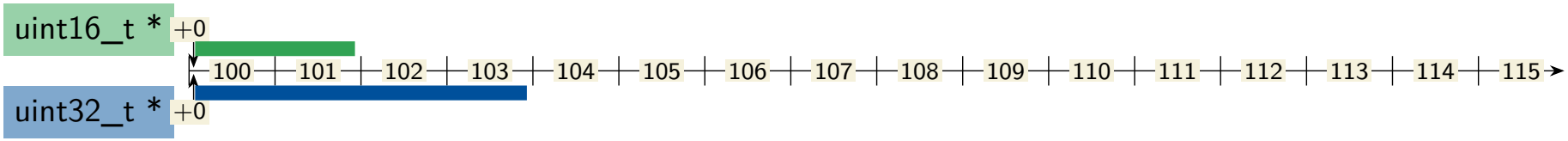
Erinnerung: Pointer sind 32/64-Bit Ganzzahlen im Speicher

- Wir können mit Pointern zu rechnen! ⇒ Pointer Arithmetric
  - Vergleiche: `!=` `<` `<=` `==` `=>` `>` , Genau wie bei Zahlen
  - Addition/Subtraktion beachten den **Pointee-Typs**

Rechnen mit Pointern

Erinnerung: Pointer sind 32/64-Bit Ganzzahlen im Speicher

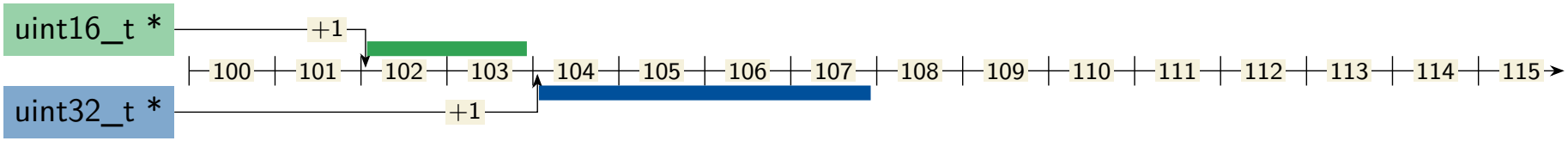
- Wir können mit Pointern zu rechnen! ⇒ Pointer Arithmetric
  - Vergleiche: `!=` `<` `<=` `==` `=>` `>` , Genau wie bei Zahlen
  - Addition/Subtraktion beachten den **Pointee-Typs**
- Addition/Subtraktion zwischen Pointer und Ganzzahl (ptr, i)



Rechnen mit Pointern

Erinnerung: Pointer sind 32/64-Bit Ganzzahlen im Speicher

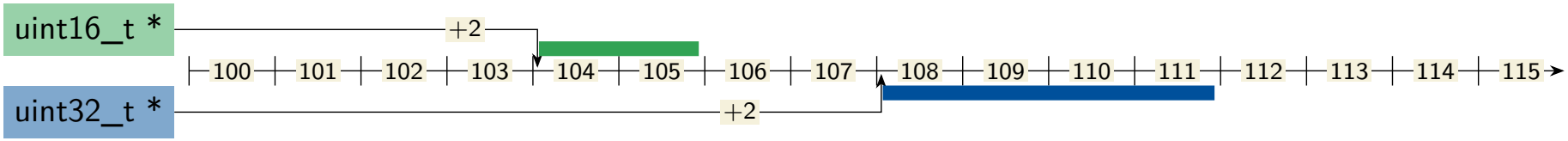
- Wir können mit Pointern zu rechnen! ⇒ Pointer Arithmetric
  - Vergleiche: `!=` `<` `<=` `==` `=>` `>` , Genau wie bei Zahlen
  - Addition/Subtraktion beachten den **Pointee-Typs**
- Addition/Subtraktion zwischen Pointer und Ganzzahl (ptr, i)
  - Die gespeicherte Adresse wird in `sizeof(*ptr)` -Schritten verändert



Rechnen mit Pointern

Erinnerung: Pointer sind 32/64-Bit Ganzzahlen im Speicher

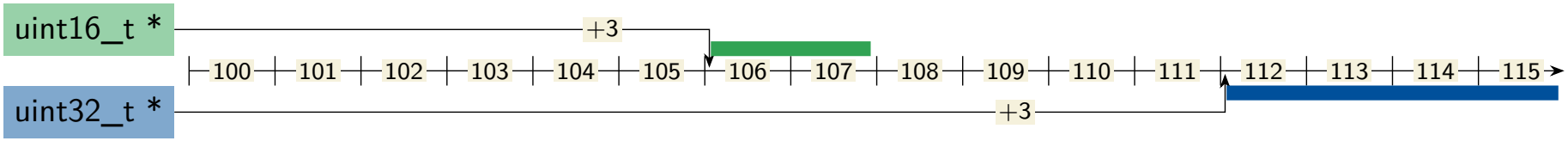
- Wir können mit Pointern zu rechnen! ⇒ Pointer Arithmetric
  - Vergleiche: `!=` `<` `<=` `==` `=>` `>` , Genau wie bei Zahlen
  - Addition/Subtraktion beachten den **Pointee-Typs**
- Addition/Subtraktion zwischen Pointer und Ganzzahl (ptr, i)
  - Die gespeicherte Adresse wird in `sizeof(*ptr)` -Schritten verändert
  - `(ptr + 1)` zeigt "hinter" die Variable auf die `ptr` zeigt.



Rechnen mit Pointern

Erinnerung: Pointer sind 32/64-Bit Ganzzahlen im Speicher

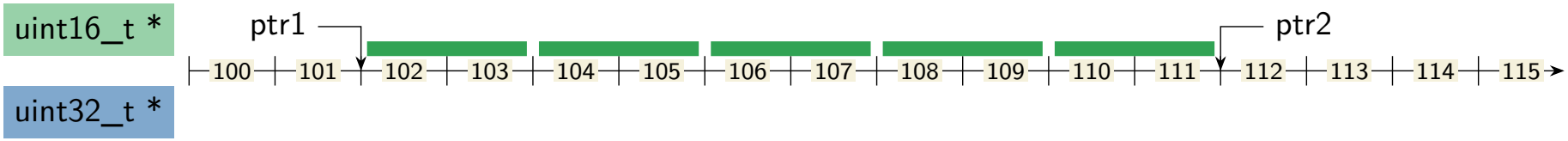
- Wir können mit Pointern zu rechnen! ⇒ Pointer Arithmetric
  - Vergleiche: `!=` `<` `<=` `==` `=>` `>` , Genau wie bei Zahlen
  - Addition/Subtraktion beachten den **Pointee-Typs**
- Addition/Subtraktion zwischen Pointer und Ganzzahl (ptr, i)
  - Die gespeicherte Adresse wird in `sizeof(*ptr)` -Schritten verändert
  - `(ptr + 1)` zeigt "hinter" die Variable auf die `ptr` zeigt.



Rechnen mit Pointern

Erinnerung: Pointer sind 32/64-Bit Ganzzahlen im Speicher

- Wir können mit Pointern zu rechnen! ⇒ Pointer Arithmetric
  - Vergleiche: `!=` `<` `<=` `==` `=>` `>` , Genau wie bei Zahlen
  - Addition/Subtraktion beachten den **Pointee-Typs**
- Addition/Subtraktion zwischen Pointer und Ganzzahl (ptr, i)
  - Die gespeicherte Adresse wird in `sizeof(*ptr)` -Schritten verändert
  - `(ptr + 1)` zeigt "hinter" die Variable auf die `ptr` zeigt.



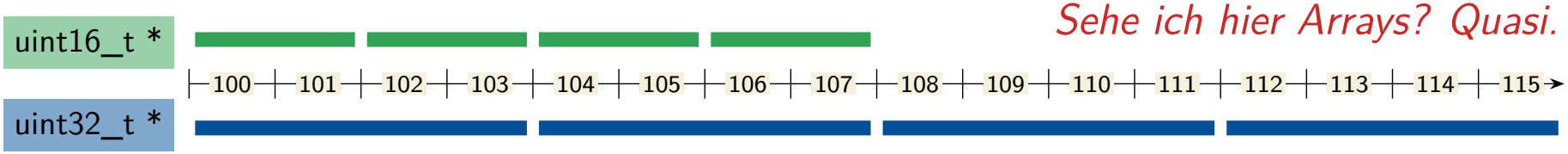
- Subtraktion zwischen Pointern gleichen Typs (ptr1, ptr2)
  - Wieviele "Elemente" liegen zwischen ptr2 und ptr1?
  - `(ptr2-ptr1)` : (Differenz der Adressen) / `sizeof(*ptr1)`

Rechnen mit Pointern

Erinnerung: Pointer sind 32/64-Bit Ganzzahlen im Speicher

- Wir können mit Pointern zu rechnen! ⇒ Pointer Arithmetric
  - Vergleiche: `!=` `<` `<=` `==` `=>` `>` , Genau wie bei Zahlen
  - Addition/Subtraktion beachten den **Pointee-Typs**
- Addition/Subtraktion zwischen Pointer und Ganzzahl (ptr, i)
  - Die gespeicherte Adresse wird in `sizeof(*ptr)` -Schritten verändert
  - `(ptr + 1)` zeigt "hinter" die Variable auf die `ptr` zeigt.

Sehe ich hier Arrays? Quasi.



- Subtraktion zwischen Pointern gleichen Typs (ptr1, ptr2)
  - Wieviele "Elemente" liegen zwischen ptr2 und ptr1?
  - `(ptr2-ptr1)` : (Differenz der Adressen) / `sizeof(*ptr1)`

Arrays in C

Erinnerung (Prog 1): Arrays sind Verbünde von Daten gleichen Typs

Syntax für Arrays

```
uint32_t arr[10]; // Deklaration
tmp    = arr[4];  // Auslesen
arr[3] = 23;      // Zuweisen
```

- 1 Array-**Variable** $\approx$ N Variablen
- 0-Indiziert
- Konsekutiv im Speicher

Arrays in C

Erinnerung (Prog 1): Arrays sind Verbünde von Daten gleichen Typs

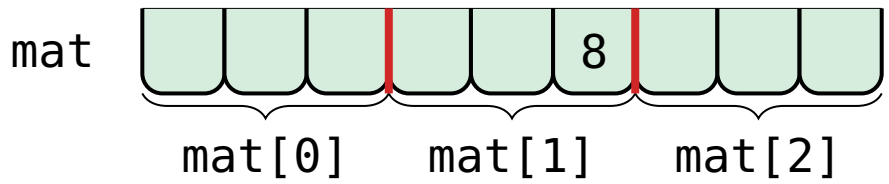
Syntax für Arrays

```
uint32_t arr[10]; // Deklaration
tmp    = arr[4];  // Auslesen
arr[3] = 23;      // Zuweisen
```

- 1 Array-**Variable** ≈ N Variablen
- 0-Indiziert
- Konsekutiv im Speicher

Mehrdimensionale Arrays

```
uint8_t mat[3][3];
mat[1][2] = 8;
```



Arrays in C

Erinnerung (Prog 1): Arrays sind Verbünde von Daten gleichen Typs

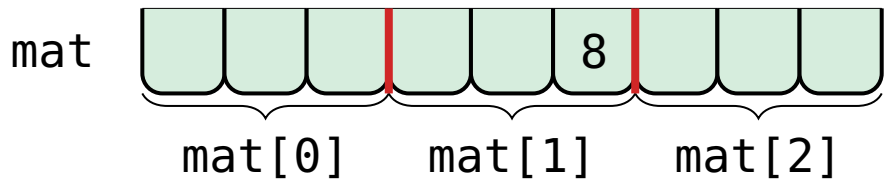
Syntax für Arrays

```
uint32_t arr[10]; // Deklaration
tmp    = arr[4];  // Auslesen
arr[3] = 23;      // Zuweisen
```

- 1 Array-**Variable** ≈ N Variablen
- 0-Indiziert
- Konsekutiv im Speicher

Mehrdimensionale Arrays

```
uint8_t mat[3][3];
mat[1][2] = 8;
```



Arrays haben Referenzsemantik

- `arr` == `&arr[0]`
- Wir bekommen einen **Pointer** und nicht den **Werteverbund**!

⇒ `arr` hat den Typ `uint32_t *`

Arrays in C

Erinnerung (Prog 1): Arrays sind Verbünde von Daten gleichen Typs

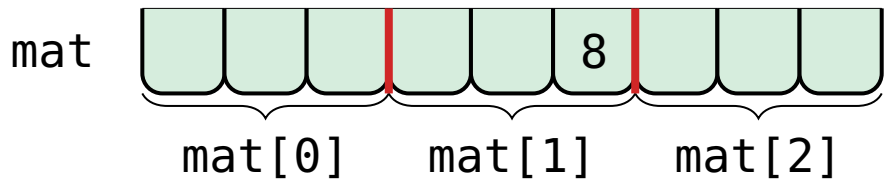
Syntax für Arrays

```
uint32_t arr[10]; // Deklaration
tmp    = arr[4];  // Auslesen
arr[3] = 23;      // Zuweisen
```

- 1 Array-**Variable** ≈ N Variablen
- 0-Indiziert
- Konsekutiv im Speicher

Mehrdimensionale Arrays

```
uint8_t mat[3][3];
mat[1][2] = 8;
```



Arrays haben Referenzsemantik

- `arr` == `&arr[0]`
- Wir bekommen einen **Pointer** und nicht den **Werteverbund**!

⇒ `arr` hat den Typ `uint32_t *`

`[]` ist nur **Syntax-Zucker**!

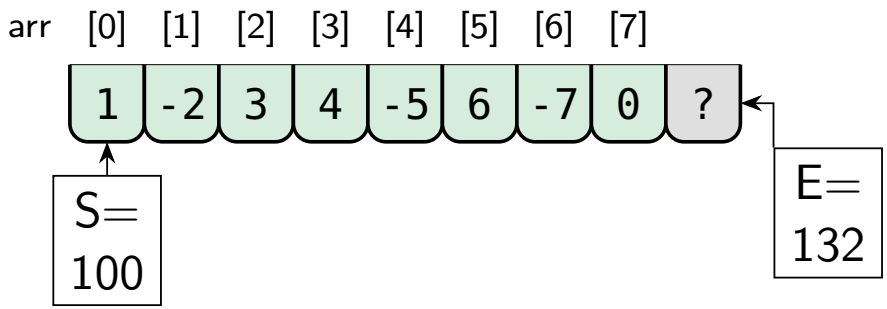
```
&(arr[0]) == arr;     // true
&(arr[5]) == arr + 5; // true

// arr[3] = arr[2];
*(arr + 3) = *(arr + 2);
```

Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

Blick in den Speicher



Konsolenausgabe

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

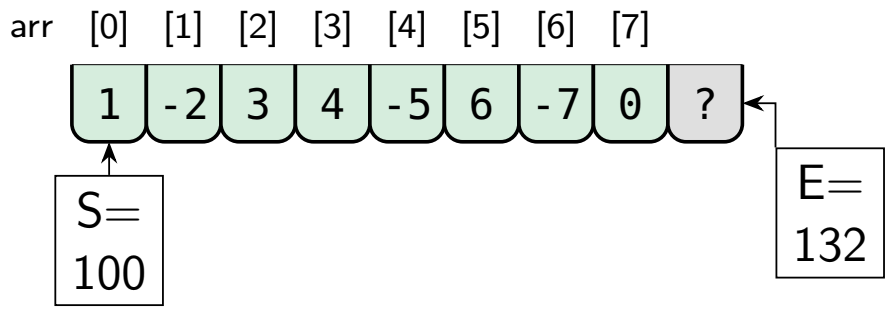
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`

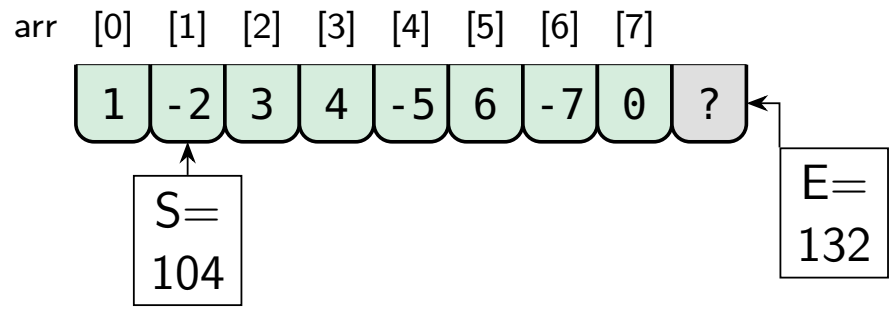
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)

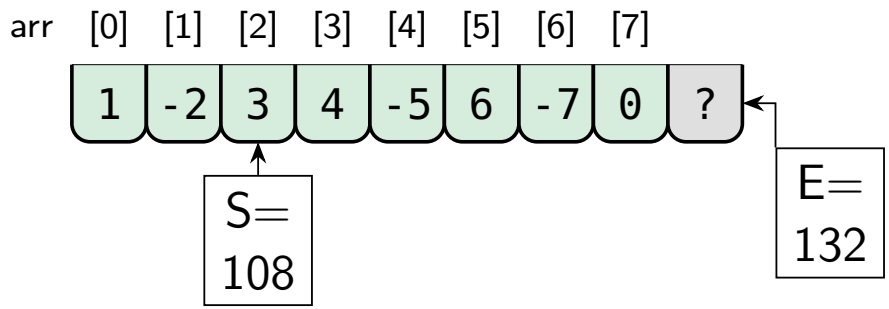
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1 -2
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)

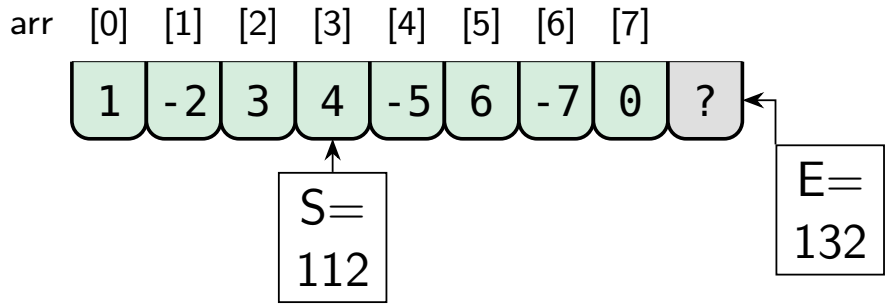
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1 -2 3
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)

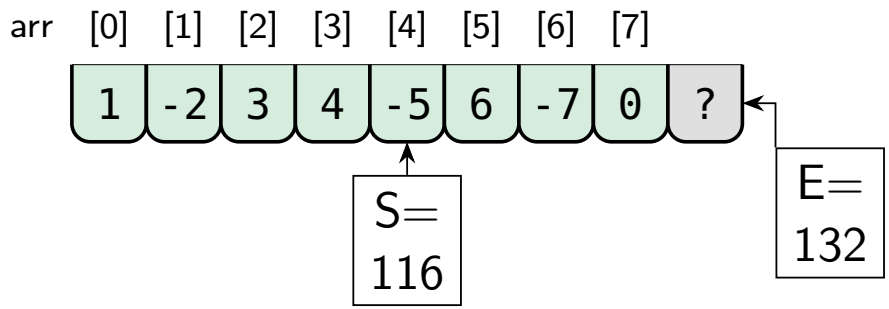
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1 -2 3 4
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)

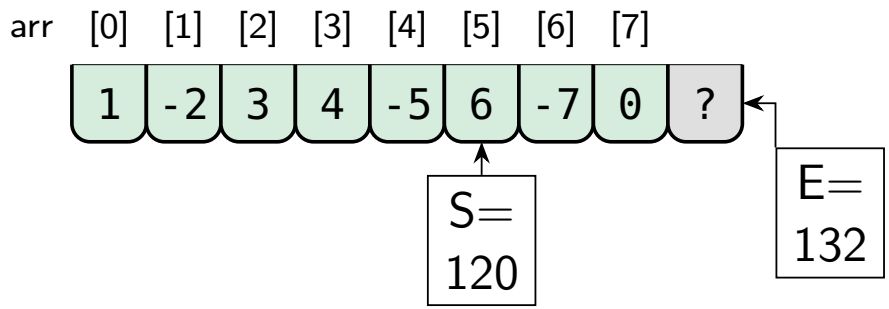
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1 -2 3 4 -5
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)

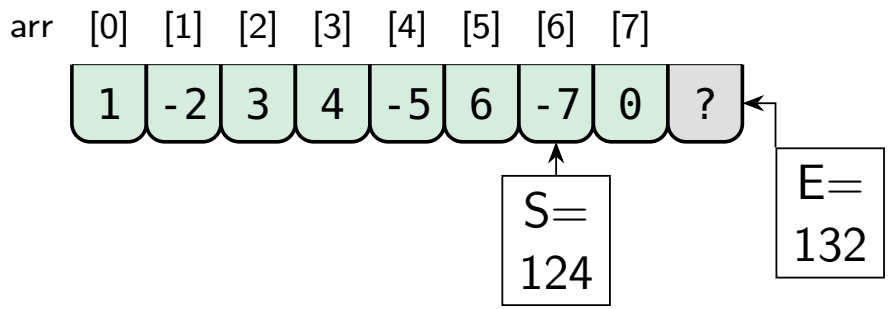
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1 -2 3 4 -5 6
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)

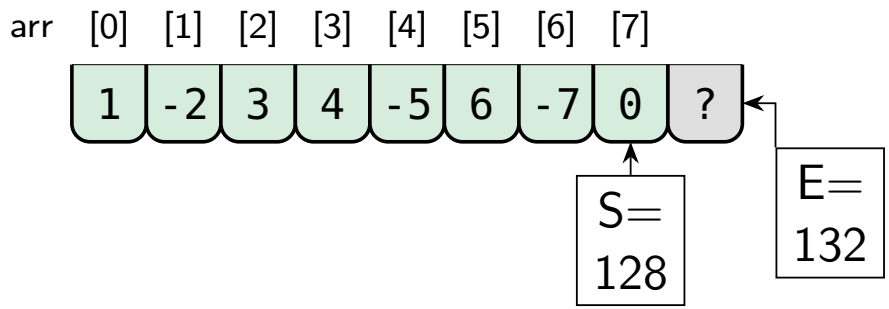
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1 -2 3 4 -5 6 -7
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)

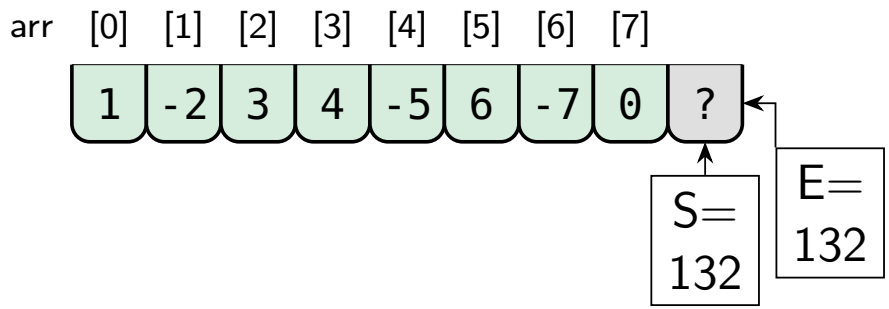
Beispiel: Ausgabe eines Arrays

Source Code

```
void dump(int *S, int *E) {
      while(S != E) {
          printf(" %d", *S);
          S = S + 1;
      }
      printf("\n");
  }

  int main() {
    int a[8]={1,-2,3,4,-5,6,-7,0};
    dump(a, a + 8);
  }
```

- main() übergibt 2 Zeiger
  - S zeigt auf a[0]
  - E zeigt **hinter** a[7]

Blick in den Speicher



Konsolenausgabe

```
1 -2 3 4 -5 6 -7 0
```

- dump(): Ausgabe von [S bis E(
  - …dereferenziert mit `*S`
  - …und inkrementiert mit (`S=S+1`)
  - …bis `S == E`, dann Abbruch