

15. Zeiger, Algorithmen, Iteratoren und Container II

Iteration mit Zeigern, Felder: Indizes vs. Zeiger, Felder und Funktionen, Zeiger und const, Algorithmen, Container und Traversierung, Vektor-Iteratoren, Typedef, Mengen, das Iterator-Konzept

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};  
for (int* p = a; p < a+5; ++p)  
    std::cout << *p << ' '; // 3 4 6 1 2
```

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};  
for (int* p = a; p < a+5; ++p)  
    std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};  
for (int* p = a; p < a+5; ++p)  
    std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};  
for (int* p = a; p < a+5; ++p)  
    std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik und Vergleiche.

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};  
for (int* p = a; p < a+5; ++p)  
    std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik und Vergleiche.
- Zeiger können dereferenziert werden.

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};  
for (int* p = a; p < a+5; ++p)  
    std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
 - Zeiger kennen Arithmetik und Vergleiche.
 - Zeiger können dereferenziert werden.
- ⇒ Mit Zeigern kann man auf Feldern operieren.

Feldargumente: *Call by (const) reference*

```
void print_vector (const int (&v) [3]) {  
    for (int i = 0; i<3 ; ++i) {  
        std::cout << v[i] << " ";  
    }  
}  
  
void make_null_vector (int (&v) [3]) {  
    for (int i = 0; i<3 ; ++i) {  
        v[i] = 0;  
    }  
}
```

Feldargumente: *Call by value*

```
void make_null_vector (int v[3]) {  
    for (int i = 0; i<3 ; ++i) {  
        v[i] = 0;  
    }  
}  
...
```

Feldargumente: *Call by value (nicht wirklich...)*

```
void make_null_vector (int v[3]) {  
    for (int i = 0; i<3 ; ++i) {  
        v[i] = 0;  
    }  
}  
...  
int a[10];  
make_null_vector (a); // setzt nur a[0], a[1], a[2]
```

Feldargumente: *Call by value (nicht wirklich...)*

```
void make_null_vector (int v[3]) {  
    for (int i = 0; i<3 ; ++i) {  
        v[i] = 0;  
    }  
}  
  
...  
int a[10];  
make_null_vector (a); // setzt nur a[0], a[1], a[2]  
  
int* b;  
make_null_vector (b); // kein Feld bei b, Crash!
```

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen $T[n]$ oder $T[]$ (Feld über T) sind äquivalent zu T^* (Zeiger auf T)

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen $T[n]$ oder $T[]$ (Feld über T) sind äquivalent zu T^* (Zeiger auf T)
- Bei der Übergabe eines Feldes wird ein Zeiger auf das erste Element übergeben

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen $T[n]$ oder $T[]$ (Feld über T) sind äquivalent zu T^* (Zeiger auf T)
- Bei der Übergabe eines Feldes wird ein Zeiger auf das erste Element übergeben
- Längeninformation geht verloren

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen $T[n]$ oder $T[]$ (Feld über T) sind äquivalent zu T^* (Zeiger auf T)
- Bei der Übergabe eines Feldes wird ein Zeiger auf das erste Element übergeben
- Längeninformation geht verloren
- Funktion kann keinen Feldausschnitt verarbeiten (Beispiel: Suche eines Elements nur im hinteren Teil des Feldes)

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Feldausschnitts

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Feldausschnitts
- *Gültiger* Bereich heisst: hier „leben“ wirklich Feldelemente

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Feldausschnitts
- *Gültiger* Bereich heisst: hier „leben“ wirklich Feldelemente
- `[begin, end)` ist leer, wenn `begin == end`

Felder in (mutierenden) Funktionen: fill

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}

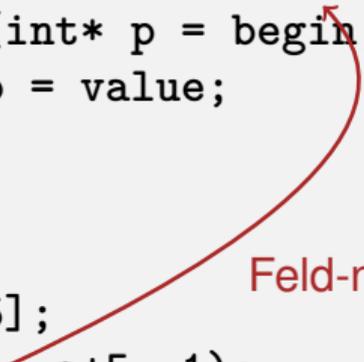
...

int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
    std::cout << a[i] << " "; // 1 1 1 1 1
```

Felder in (mutierenden) Funktionen: fill

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}
...
```

Feld-nach-Zeiger-Konversion



```
int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
    std::cout << a[i] << " "; // 1 1 1 1 1
```

Felder in (mutierenden) Funktionen: fill

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}
...
```

Erwartet Zeiger auf das erste Element
eines Bereichs

```
int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
    std::cout << a[i] << " "; // 1 1 1 1 1
```

Felder in (mutierenden) Funktionen: fill

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}
...
```

Erwartet Zeiger auf das erste Element eines Bereichs

```
int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
    std::cout << a[i] << " ";
```

Übergabe der Adresse (des ersten Elements) von a

Funktionen mit/ohne Effekten

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden. Beispiel: `fill`

Funktionen mit/ohne Effekten

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden. Beispiel: `fill`
- Aber viele Funktionen haben keinen Effekt, sie lesen Daten nur
- \Rightarrow Verwendung von `const`

Funktionen mit/ohne Effekten

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden. Beispiel: `fill`
- Aber viele Funktionen haben keinen Effekt, sie lesen Daten nur
- \Rightarrow Verwendung von `const`

Bisher, zum Beispiel:

```
int i = 0;  
const int& j = i;
```

Funktionen mit/ohne Effekten

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden. Beispiel: `fill`
- Aber viele Funktionen haben keinen Effekt, sie lesen Daten nur
- \Rightarrow Verwendung von `const`

Bisher, zum Beispiel:

```
int i = 0;
const int& j = i;

const int zero = 0;
const int& nil = zero;
```

Positionierung von Const

const T ist äquivalent zu *T const* und kann auch so geschrieben werden

```
const int zero = ...  $\iff$  int const zero = ...  
const int& nil = ...  $\iff$  int const& nil = ...
```

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const a;
```

a ist eine konstante Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const a;
```

a ist eine konstante Ganzzahl

```
int const* a;
```

a ist ein Zeiger auf eine konstante Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const a;
```

a ist eine konstante Ganzzahl

```
int const* a;
```

a ist ein Zeiger auf eine konstante Ganzzahl

```
int* const a;
```

a ist ein konstanter Zeiger auf eine Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

<code>int const a;</code>	a ist eine konstante Ganzzahl
<code>int const* a;</code>	a ist ein Zeiger auf eine konstante Ganzzahl
<code>int* const a;</code>	a ist ein konstanter Zeiger auf eine Ganzzahl
<code>int const* const a;</code>	a ist ein konstanter Zeiger auf eine konstante Ganzzahl

Nicht-mutierende Funktionen: `min`

```
// PRE: [begin, end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min (int* begin, int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for (int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```

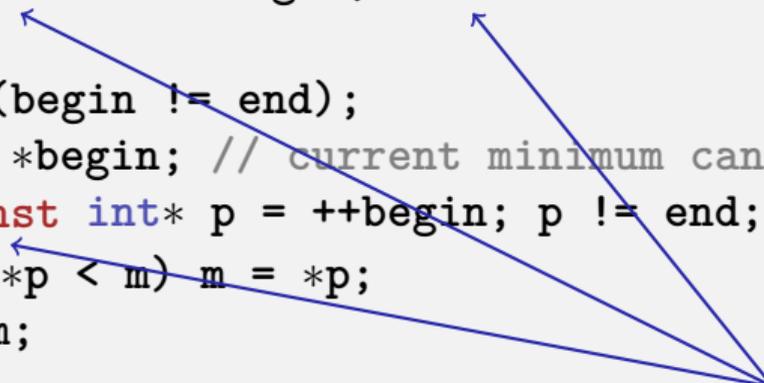
Nicht-mutierende Funktionen: `min`

```
// PRE: [begin, end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min (int* begin, int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for (int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```

- Kennzeichnung mit `const`

Nicht-mutierende Funktionen: min

```
// PRE: [begin, end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min (const int* begin, const int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for (const int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```

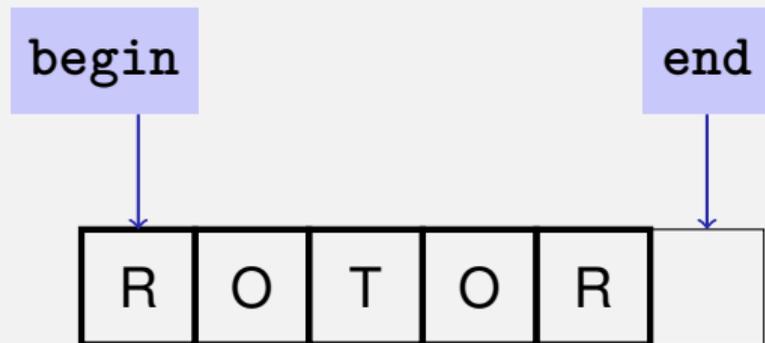
The diagram consists of three blue arrows pointing from a text box to the 'const' keywords in the code. One arrow points to 'const int*' in the function signature, another points to 'const int*' in the for loop, and a third points to 'const int*' in the if statement.

const bezieht sich auf int,
nicht auf den Zeiger.

- Kennzeichnung mit const

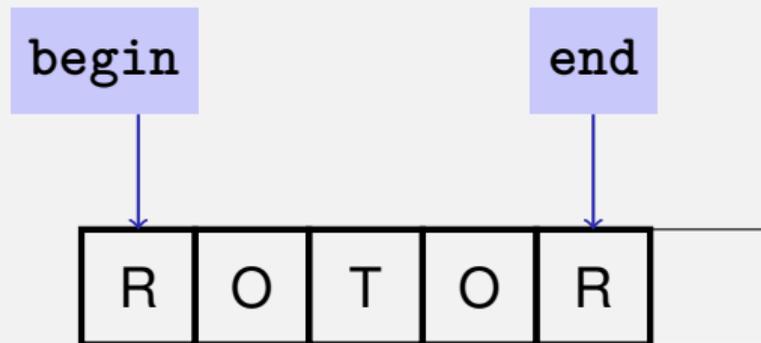
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



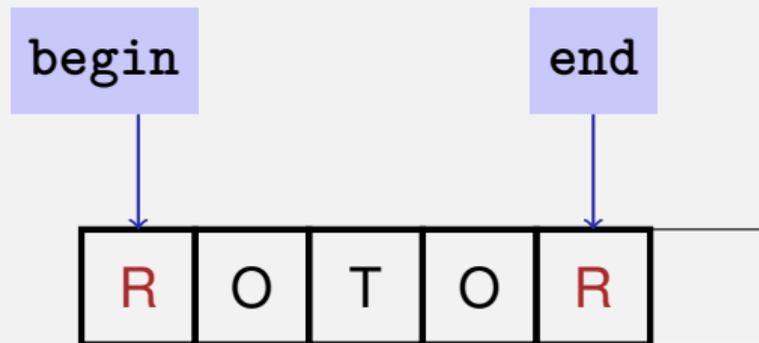
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



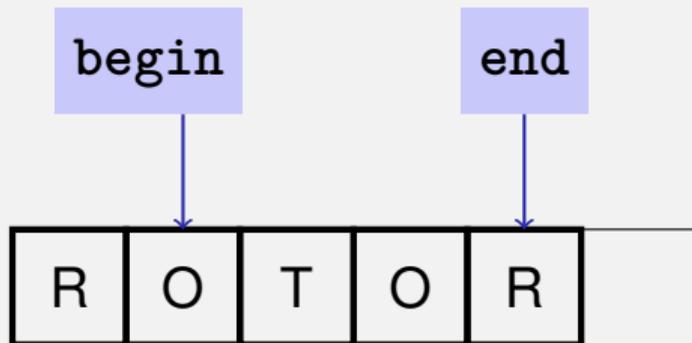
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



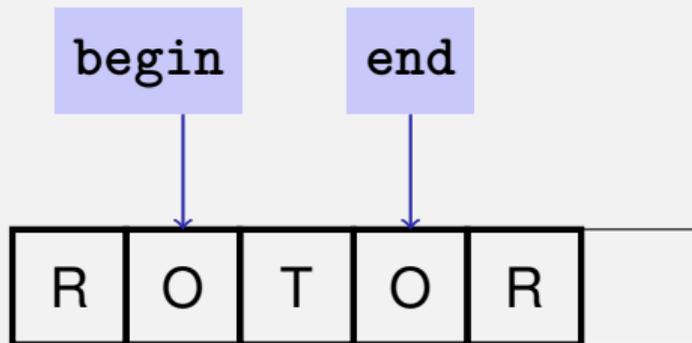
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



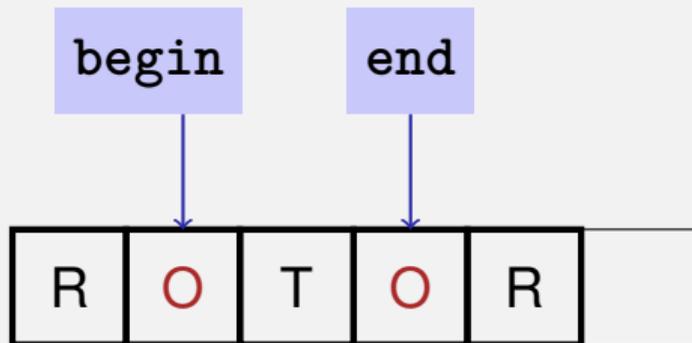
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



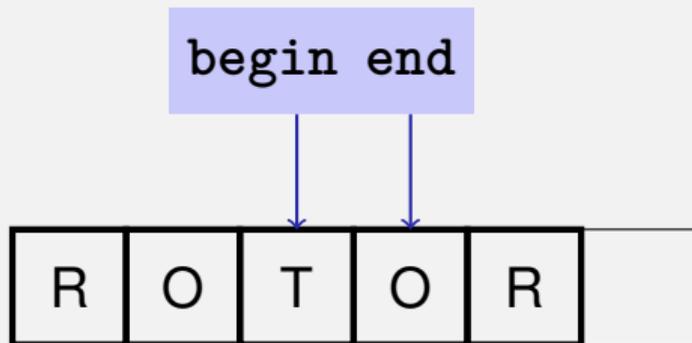
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```

begin == end



Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```

begin == end



Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```

begin == end



Algorithmen

Für viele alltägliche Probleme existieren vorgefertigte Lösungen in der Standardbibliothek.

Beispiel: Füllen eines Feldes

```
#include <algorithm> // needed for std::fill
...

int a[5];
std::fill (a, a+5, 1);

for (int i=0; i<5; ++i)
    std::cout << a[i] << " "; // 1 1 1 1 1
```

Algorithmen

Die gleichen vorgefertigten Algorithmen funktionieren für viele verschiedene Datentypen.

Beispiel: Füllen eines Feldes

```
#include <algorithm> // needed for std::fill
...

char c[3];
std::fill (c, c+3, '!');

for (int i=0; i<3; ++i)
    std::cout << c[i]; // !!!
```

Exkurs: Templates

Beispiel: fill mit Templates

```
template <typename T>
void fill (T* begin, T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}

int a[5];
fill (a, a+5, 1);    // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

Exkurs: Templates

Beispiel: fill mit Templates

```
template <typename T>
void fill(T* begin, T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}

int a[5];
fill (a, a+5, 1);    // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

Die eckigen Klammern kennen wir schon von `std::vector<int>`. Vektoren sind auch als Templates realisiert.

Exkurs: Templates

Beispiel: fill mit Templates

```
template <typename T>
void fill(T* begin, T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}
int a[5];
fill (a, a+5, 1);    // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

Die eckigen Klammern kennen wir schon von `std::vector<int>`. Vektoren sind auch als Templates realisiert.

Auch `std::fill` ist als Template realisiert!

Container und Traversierung

- **Container:** Behälter (Feld, Vektor, . . .) für Elemente

Container und Traversierung

- **Container:** Behälter (Feld, Vektor, . . .) für Elemente
- **Traversierung:** Durchlaufen eines Containers

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
 - Initialisierung der Elemente (`fill`)

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
 - Initialisierung der Elemente (`fill`)
 - Suchen des kleinsten Elements (`min`)

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
 - Initialisierung der Elemente (`fill`)
 - Suchen des kleinsten Elements (`min`)
 - Prüfen von Eigenschaften (`is_palindrome`)

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
 - Initialisierung der Elemente (`fill`)
 - Suchen des kleinsten Elements (`min`)
 - Prüfen von Eigenschaften (`is_palindrome`)
 - ...

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
 - Initialisierung der Elemente (`fill`)
 - Suchen des kleinsten Elements (`min`)
 - Prüfen von Eigenschaften (`is_palindrome`)
 - ...
- Es gibt noch viele andere Container (Mengen, Listen,...)

Werkzeuge zur Traversierung

- Felder: Indizes (wahlfrei) oder Zeiger (sequenziell)

Werkzeuge zur Traversierung

- Felder: Indizes (wahlfrei) oder Zeiger (sequenziell)
- Feld-Algorithmen (`std::`) benutzen Zeiger

```
int a[5];  
std::fill (a, a+5, 1); // 1 1 1 1 1
```

Werkzeuge zur Traversierung

- Felder: Indizes (wahlfrei) oder Zeiger (sequenziell)
- Feld-Algorithmen (`std::`) benutzen Zeiger

```
int a[5];  
std::fill (a, a+5, 1); // 1 1 1 1 1
```

- Wie traversiert man Vektoren und andere Container?

```
std::vector<int> v (5, 0); // 0 0 0 0 0  
std::fill (?, ?, 1); // 1 1 1 1 1
```

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht. . .

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);  
std::fill (v, v+5, 1); // Fehlermeldung des Compilers !
```

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);  
std::fill (v, v+5, 1); // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);  
std::fill (v, v+5, 1); // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

- Sie lassen sich weder in Zeiger konvertieren,...

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);  
std::fill (v, v+5, 1); // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

- Sie lassen sich weder in Zeiger konvertieren,...
- ...noch mit Zeigern traversieren.

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

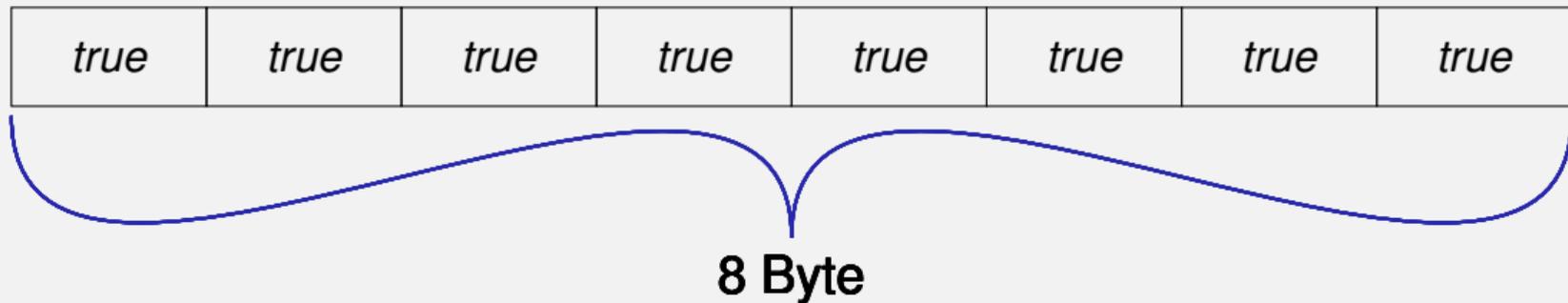
```
std::vector<int> v (5, 0);  
std::fill (v, v+5, 1); // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

- Sie lassen sich weder in Zeiger konvertieren,...
- ...noch mit Zeigern traversieren.
- Das ist ihnen viel zu primitiv. 😊

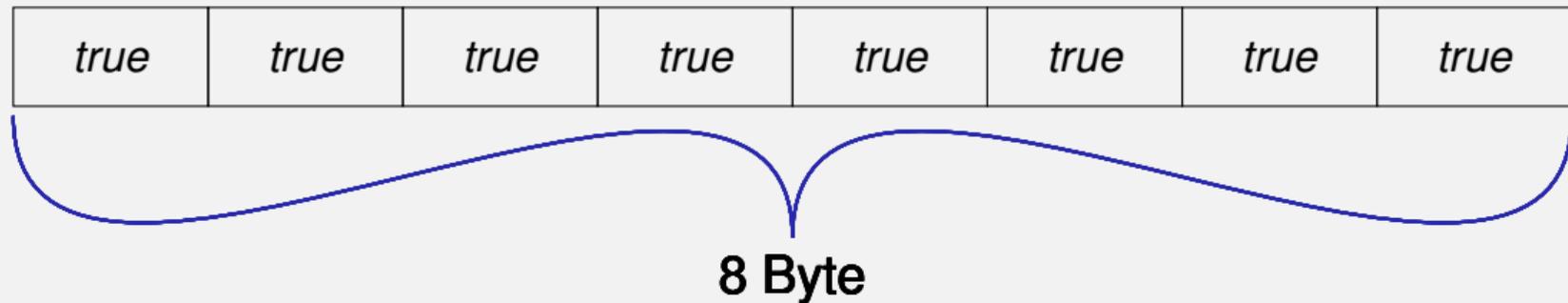
Auch im Speicher: Vektor \neq Feld

```
bool a[8] = {true, true, true, true, true, true, true, true};
```



Auch im Speicher: Vektor \neq Feld

```
bool a[8] = {true, true, true, true, true, true, true, true};
```

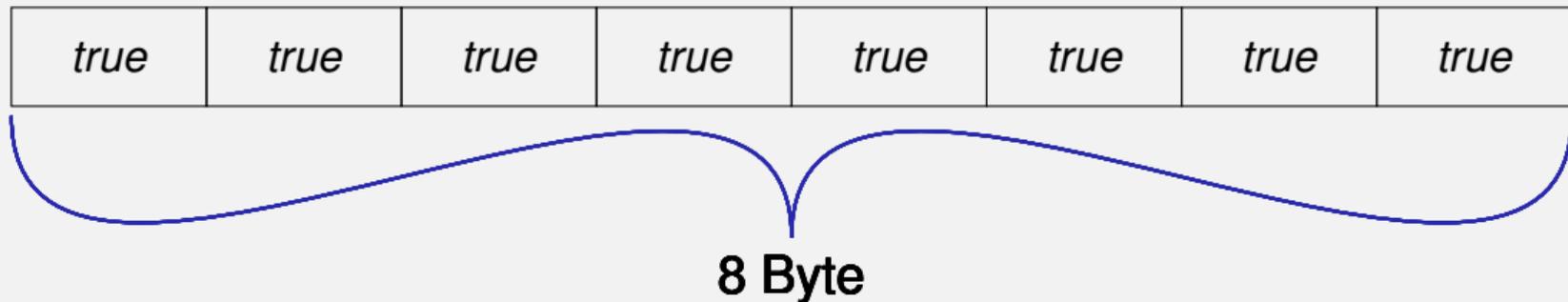


```
std::vector<bool> v (8, true);
```

0b11111111 1 Byte

Auch im Speicher: Vektor \neq Feld

```
bool a[8] = {true, true, true, true, true, true, true, true};
```



```
std::vector<bool> v (8, true);
```

`0b11111111` 1 Byte

`bool*`-Zeiger passt hier nicht, denn er läuft **byte**weise, nicht **bit**weise!

Vektor-Iteratoren

Iterator: ein „Zeiger“, der zum Container passt.

Vektor-Iteratoren

Iterator: ein „Zeiger“, der zum Container passt.

Beispiel: Füllen eines Vektors mit `std::fill` – so geht's!

```
#include <vector>
#include <algorithm> // needed for std::fill

...
std::vector<int> v(5, 0);
std::fill (v.begin(), v.end(), 1);
for (int i=0; i<5; ++i)
    std::cout << v[i] << " "; // 1 1 1 1 1
```

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`
 - für nicht-mutierenden Zugriff

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`

- für nicht-mutierenden Zugriff
- analog zu `const int*` für Felder

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`

- für nicht-mutierenden Zugriff
- analog zu `const int*` für Felder

- `std::vector<int>::iterator`

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

■ `std::vector<int>::const_iterator`

- für nicht-mutierenden Zugriff
- analog zu `const int*` für Felder

■ `std::vector<int>::iterator`

- für mutierenden Zugriff

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

■ `std::vector<int>::const_iterator`

- für nicht-mutierenden Zugriff
- analog zu `const int*` für Felder

■ `std::vector<int>::iterator`

- für mutierenden Zugriff
- analog zu `int*` für Felder

Vektor-Iteratoren: `begin()` und `end()`

- `v.begin()` zeigt auf das erste Element von `v`
- `v.end()` zeigt hinter das letzte Element von `v`

Vektor-Iteratoren: `begin()` und `end()`

- `v.begin()` zeigt auf das erste Element von `v`
- `v.end()` zeigt hinter das letzte Element von `v`

Vektor-Iteratoren: begin() und end()

- Damit können wir einen Vektor traversieren...

```
for (std::vector<int>::const_iterator it = v.begin();  
     it != v.end(); ++it)  
    std::cout << *it << " ";
```

- ...oder einen Vektor füllen.

```
std::fill (v.begin(), v.end(), 1);
```

Vektor-Iteratoren: begin() und end()

- Damit können wir einen Vektor traversieren...

```
for (std::vector<int>::const_iterator it = v.begin();  
     it != v.end(); ++it)  
    std::cout << *it << " ";
```

- ...oder einen Vektor füllen.

```
std::fill (v.begin(), v.end(), 1);
```

Typnamen in C++ können laaaaaaang werden

- `std::vector<int>::const_iterator`

Typnamen in C++ können laaaaaaang werden

- `std::vector<int>::const_iterator`
- Dann hilft die Deklaration eines *Typ-Alias* mit

`using Name = Typ;`

Name, unter dem der Typ neu
auch angesprochen werden kann

bestehender Typ

Typnamen in C++ können laaaaaaang werden

- `std::vector<int>::const_iterator`
- Dann hilft die Deklaration eines *Typ-Alias* mit

`using Name = Typ;`

Name, unter dem der Typ neu
auch angesprochen werden kann

bestehender Typ

Beispiele

```
using int_vec = std::vector<int>;  
using Cvit = int_vec::const_iterator;
```

Vektor-Iteratoren funktionieren wie Zeiger

```
using Cvit = std::vector<int>::const_iterator;

std::vector<int> v(5, 0); // 0 0 0 0 0

// output all elements of a, using iteration
for (Cvit it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
```

Vektor-Iteratoren funktionieren wie Zeiger

```
using Cvit = std::vector<int>::const_iterator;
```

```
std::vector<int> v(5, 0); // 0 0 0 0 0
```

```
// output all elements of a, using iteration  
for (Cvit it = v.begin(); it != v.end(); ++it)  
    std::cout << *it << " ";
```



Vektor-Element,
auf das it zeigt

Vektor-Iteratoren funktionieren wie Zeiger

```
using Vit = std::vector<int>::iterator;

// manually set all elements to 1
for (Vit it = v.begin(); it != v.end(); ++it)
    *it = 1;

// output all elements again, using random access
for (int i=0; i<5; ++i)
    std::cout << v[i] << " ";
```

Vektor-Iteratoren funktionieren wie Zeiger

```
using Vit = std::vector<int>::iterator;
```

```
// manually set all elements to 1
```

```
for (Vit it = v.begin(); it != v.end(); ++it)
```

```
    *it = 1;
```

Inkrementieren des Iterators



```
// output all elements again, using random access
```

```
for (int i=0; i<5; ++i)
```

```
    std::cout << v[i] << " ";
```

Kurzschreibweise für
*(v.begin()+i)



Andere Container: Mengen (Sets)

- Eine Menge ist eine ungeordnete Zusammenfassung von Elementen, wobei jedes Element nur einmal vorkommt.

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

Andere Container: Mengen (Sets)

- Eine Menge ist eine ungeordnete Zusammenfassung von Elementen, wobei jedes Element nur einmal vorkommt.

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- C++: `std::set<T>` für eine Menge mit Elementen vom Typ T

Mengen: Beispiel einer Anwendung

- Stelle fest, ob ein gegebener Text ein Fragezeichen enthält und gib alle im Text vorkommenden *verschiedenen* Zeichen aus!

Buchstabensalat (1)

```
#include<set>
...
using Csit = std::set<char>::const_iterator;
...
std::string text =
    "What are the distinct characters in this string?";

std::set<char> s (text.begin(),text.end());
```

Buchstabensalat (1)

```
#include<set>
...
using Csit = std::set<char>::const_iterator;
...
std::string text =
    "What are the distinct characters in this string?";

std::set<char> s (text.begin(),text.end());
```



Menge wird mit *String-Iterator-Bereich*
[text.begin(), text.end()) initialisiert

Buchstabensalat (2)

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Buchstabensalat (2)

Suchalgorithmus, aufrufbar mit beliebigem
Iterator-Bereich

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Buchstabensalat (2)

Suchalgorithmus, aufrufbar mit beliebigem
Iterator-Bereich

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";
```

```
// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Ausgabe:
Good question!
?Wacdeghinrst

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren?

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren?

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren?

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

Fehlermeldung: no subscript operator

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren? **Nein.**

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

Fehlermeldung: no subscript operator

- Mengen sind ungeordnet.
 - Es gibt kein „*i*-tes Element“.

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren? **Nein.**

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

Fehlermeldung: no subscript operator

- Mengen sind ungeordnet.
 - Es gibt kein „*i*-tes Element“.
 - Iteratorvergleich `it != s.end()` geht, nicht aber `it < s.end()`!

Das Konzept der Iteratoren

C++ kennt verschiedene Iterator-Typen

Das Konzept der Iteratoren

C++ kennt verschiedene Iterator-Typen

- Jeder Container hat einen zugehörigen Iterator-Typ

Das Konzept der Iteratoren

C++ kennt verschiedene Iterator-Typen

- Jeder Container hat einen zugehörigen Iterator-Typ
- Alle können dereferenzieren (`*it`) und traversieren (`++it`)

Das Konzept der Iteratoren

C++ kennt verschiedene Iterator-Typen

- Jeder Container hat einen zugehörigen Iterator-Typ
- Alle können dereferenzieren (`*it`) und traversieren (`++it`)
- Manche können mehr, z.B. wahlfreien Zugriff (`it[k]`, oder äquivalent `*(it + k)`), rückwärts traversieren (`--it`),...

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*.
Das heisst:

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*.
Das heisst:

- Der Container wird per Iterator-Bereich übergeben

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*.
Das heisst:

- Der Container wird per Iterator-Bereich übergeben
- Der Algorithmus funktioniert für alle Container, deren Iteratoren die Anforderungen des Algorithmus erfüllen

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*.
Das heisst:

- Der Container wird per Iterator-Bereich übergeben
- Der Algorithmus funktioniert für alle Container, deren Iteratoren die Anforderungen des Algorithmus erfüllen
- `std::find` erfordert z.B. nur `*` und `++`

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*.
Das heisst:

- Der Container wird per Iterator-Bereich übergeben
- Der Algorithmus funktioniert für alle Container, deren Iteratoren die Anforderungen des Algorithmus erfüllen
- `std::find` erfordert z.B. nur `*` und `++`
- Implementationsdetails des Containers sind nicht von Bedeutung

16. Rekursion 1

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich **rekursiv** definierbar.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich **rekursiv** definierbar.
- Das heisst, die Funktion erscheint in ihrer eigenen Definition.

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n - 1)!, & \text{andernfalls} \end{cases}$$

Rekursion in C++: Genauso!

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n - 1)!, & \text{andernfalls} \end{cases}$$

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac (n-1);  
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife. . .

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife...
- ... nur noch schlechter ("verbrennt" Zeit **und** Speicher)

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife...
- ... nur noch schlechter ("verbrennt" Zeit **und** Speicher)

```
void f()  
{  
    f(); // f() -> f() -> ... stack overflow  
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife...
- ...nur noch schlechter ("verbrennt" Zeit **und** Speicher)

```
void f()  
{  
    f(); // f() -> f() -> ... stack overflow  
}
```

Ein Euro ist ein Euro.

Wim Duisenberg, erster Präsident der EZB

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)` :

terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument $< n$ aufgerufen.

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)` :

terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument $< n$ aufgerufen.

„n wird mit jedem Aufruf kleiner.“

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{  
    if (n <= 1) return 1;  
    return n * fac(n-1); // n > 1  
}
```

Aufruf von `fac(4)`

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{ // n = 4  
    if (n <= 1) return 1;  
    return n * fac(n-1); // n > 1  
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{ // n = 4  
    if (n <= 1) return 1;  
    return n * fac(n-1); // n > 1  
}
```

Auswertung des Rückgabedruckes

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{ // n = 4  
    if (n <= 1) return 1;  
    return n * fac(n-1); // n > 1  
}
```

Rekursiver Aufruf mit Argument $n - 1 == 3$

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{ // n = 3  
    if (n <= 1) return 1;  
    return n * fac(n-1); // n > 1  
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{ // n = 3  
    if (n <= 1) return 1;  
    return n * fac(n-1); // n > 1  
}
```

Es gibt jetzt zwei n . Das von `fac(4)` und das von `fac(3)`

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!  
unsigned int fac (unsigned int n)  
{  
    if (n <= 1) return 1;  
    return n * fac(n-1); // n > 1  
}
```

Es wird mit dem n des aktuellen Aufrufs gearbeitet: $n = 3$

Initialisierung des formalen Arguments

Der Aufrufstapel

```
std::cout << fac(4)
```

Der Aufrufstapel

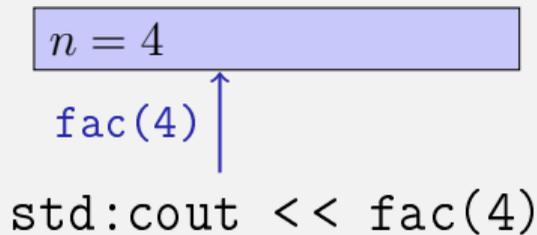
Bei jedem Funktionsaufruf:

```
    fac(4) ↑  
std::cout << fac(4)
```

Der Aufrufstapel

Bei jedem Funktionsaufruf:

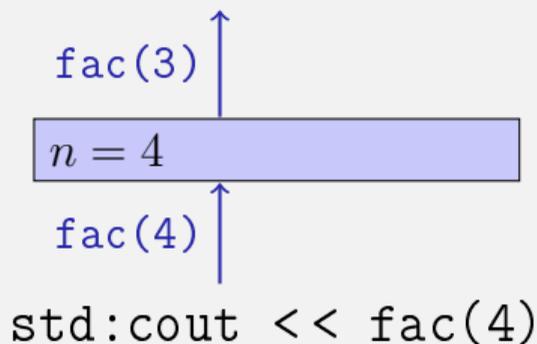
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

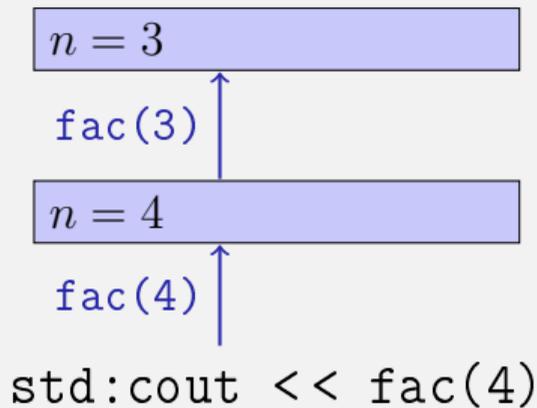
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

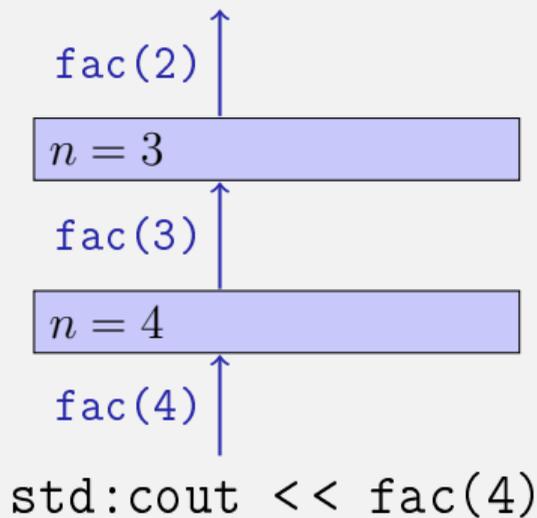
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

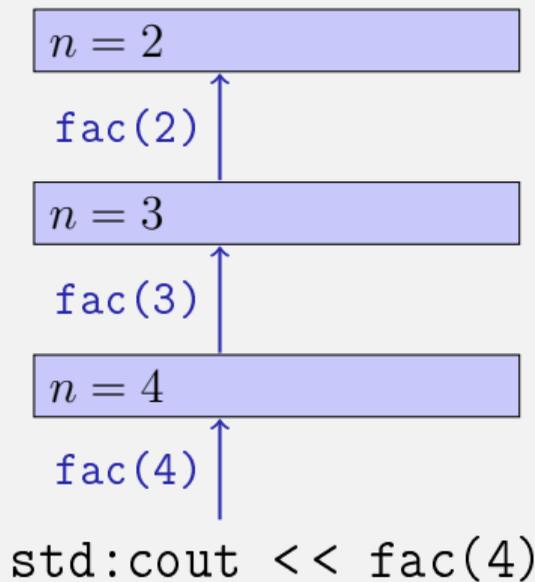
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

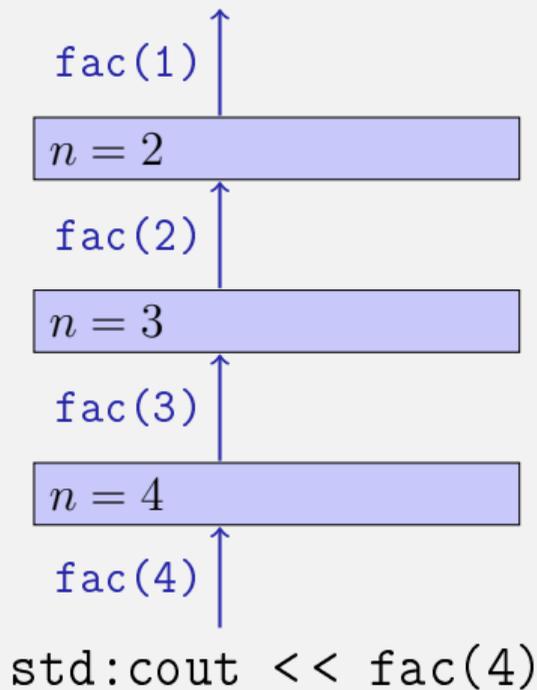
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

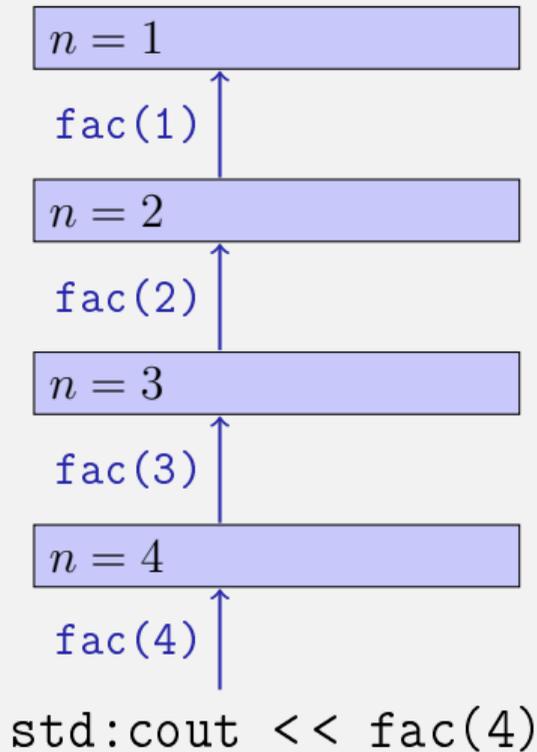
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

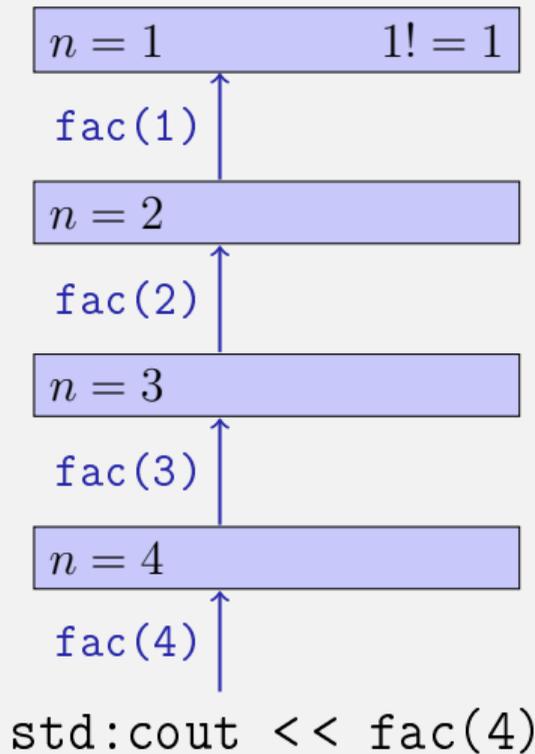
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

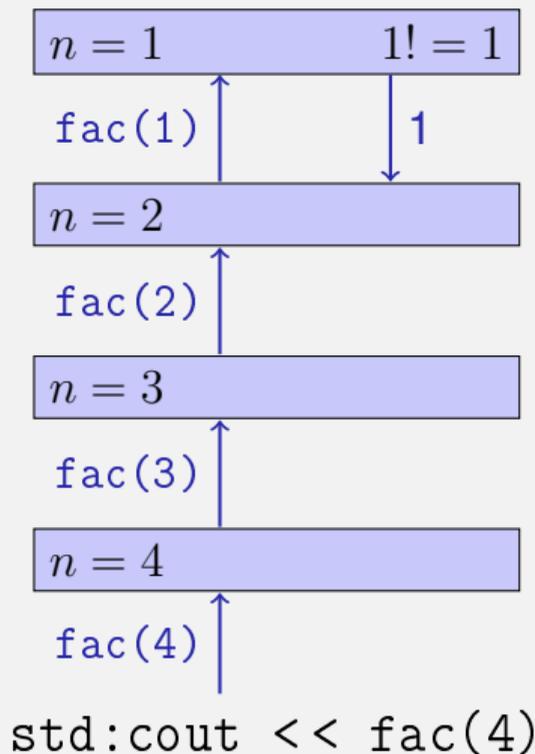
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet



Der Aufrufstapel

Bei jedem Funktionsaufruf:

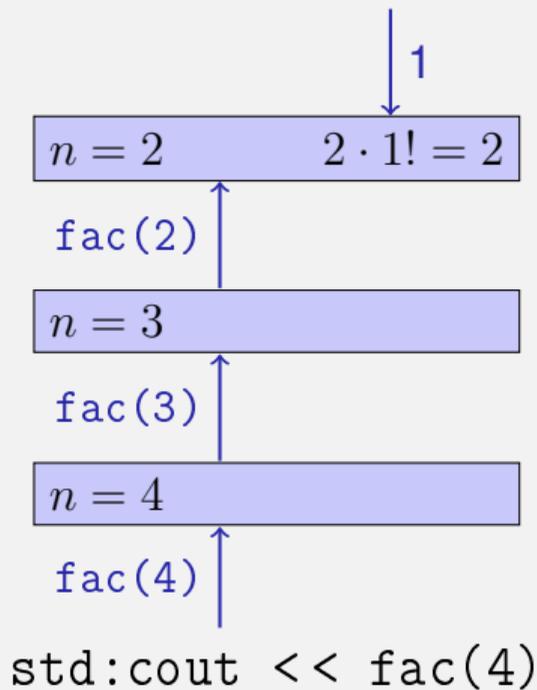
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

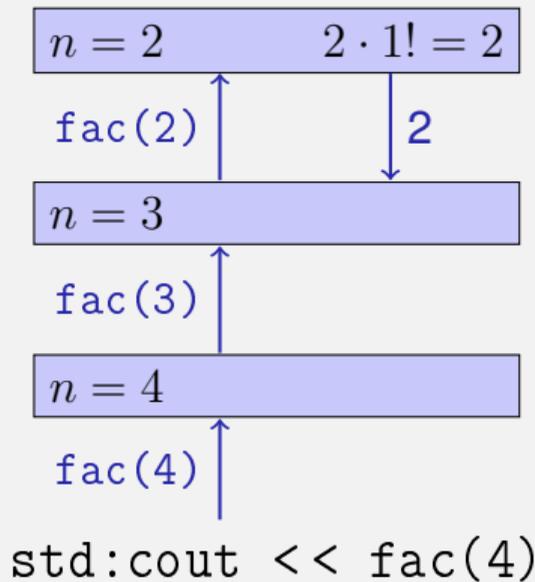
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

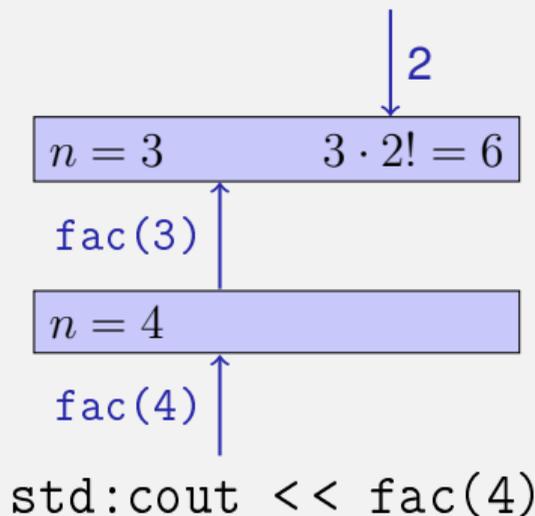
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

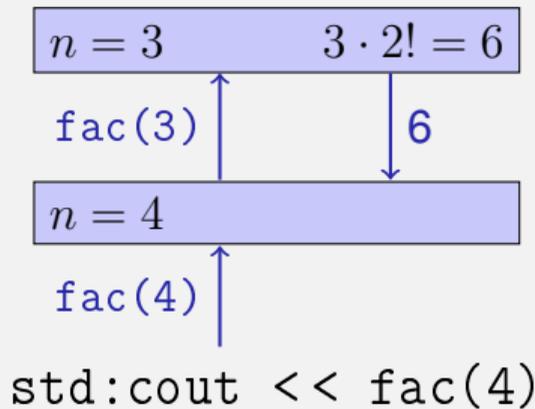
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

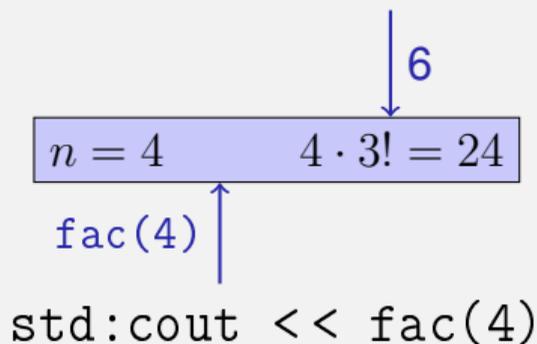
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

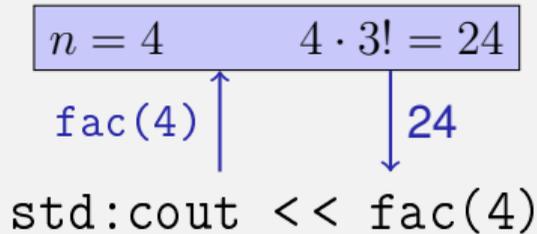
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht

`std::cout << fac(4)`



A blue arrow points downwards from the number 24 to the closing parenthesis of the function call fac(4) in the code line above.

Die Macht der Rekursion

- Einige Probleme scheinen ohne Rekursion kaum lösbar zu sein. Mit Rekursion werden sie plötzlich einfacher lösbar.
- Beispiele: *Die Türme von Hanoi*, das n -Damen-Problem, Parsen von Ausdrücken, *Sudoku-Löser*, Umgekehrte Aus- oder Eingabe, Suchen in Bäumen, Divide-And-Conquer (z.B. Sortieren) → Engineering Tool III-IV

Experiment: Die Türme von Hanoi



Links

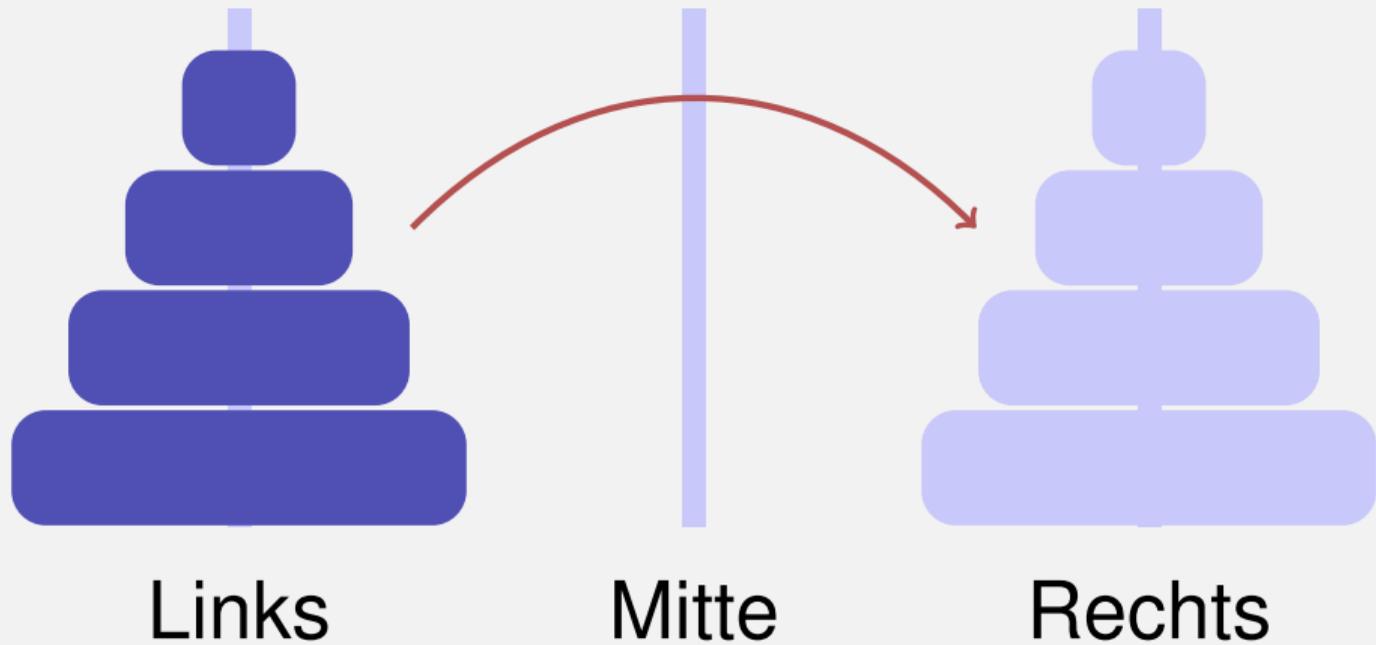


Mitte



Rechts

Experiment: Die Türme von Hanoi



Die Türme von Hanoi - So gehts!



Links



Mitte



Rechts

Die Türme von Hanoi - So gehts!



Links

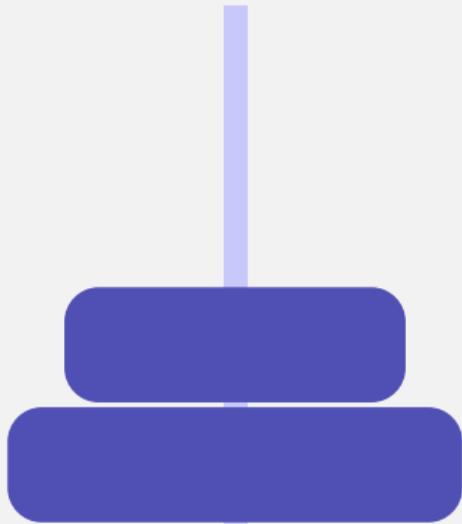


Mitte



Rechts

Die Türme von Hanoi - So gehts!



Links



Mitte



Rechts

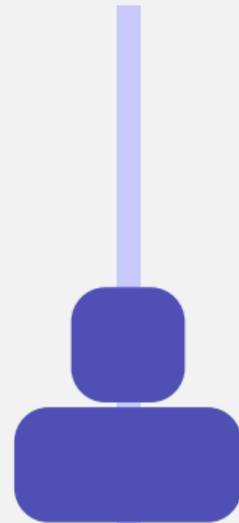
Die Türme von Hanoi - So gehts!



Links

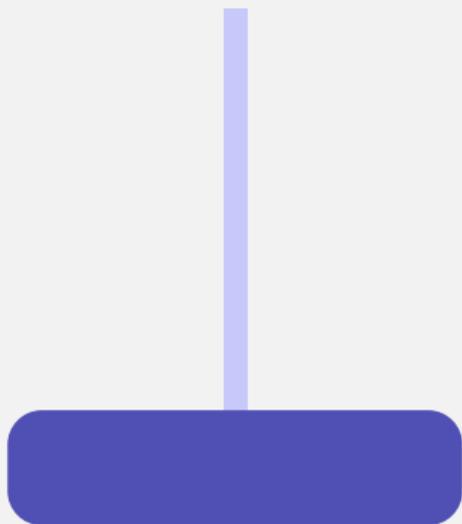


Mitte

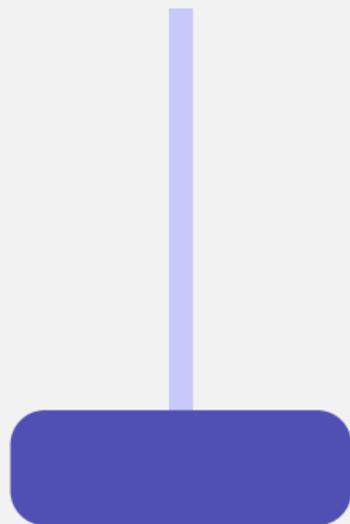


Rechts

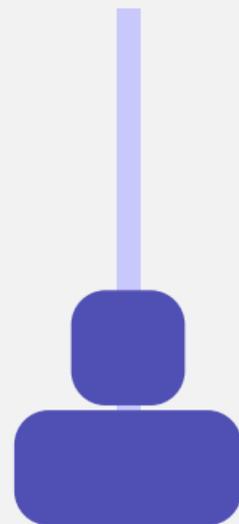
Die Türme von Hanoi - So gehts!



Links

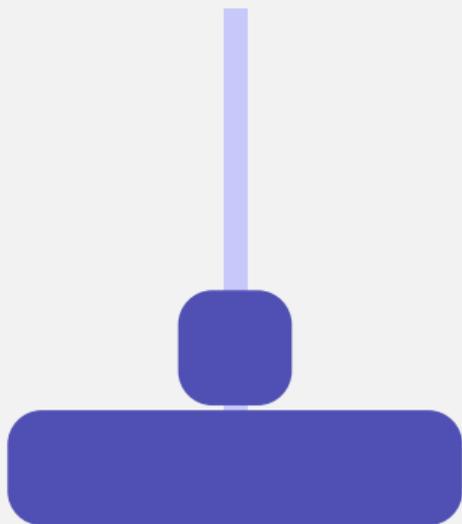


Mitte

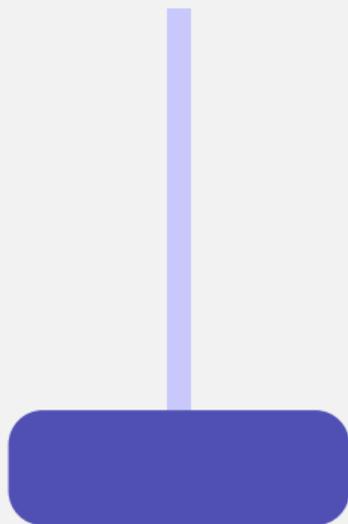


Rechts

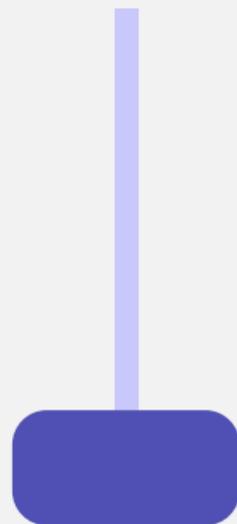
Die Türme von Hanoi - So gehts!



Links

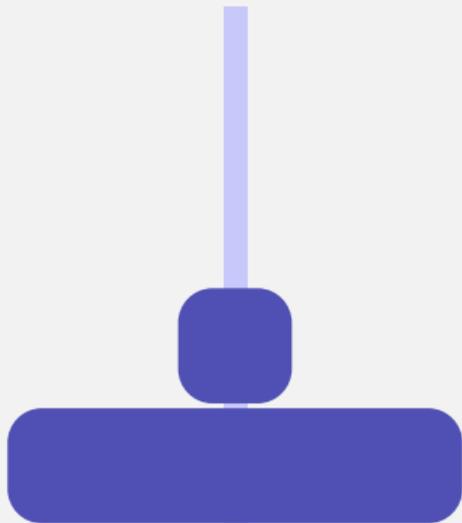


Mitte

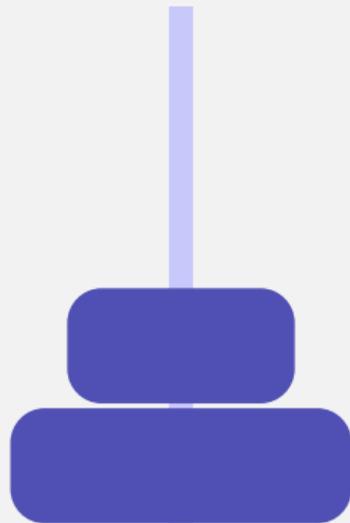


Rechts

Die Türme von Hanoi - So gehts!



Links

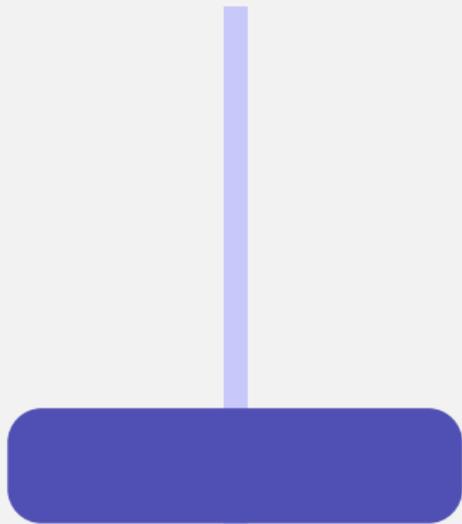


Mitte

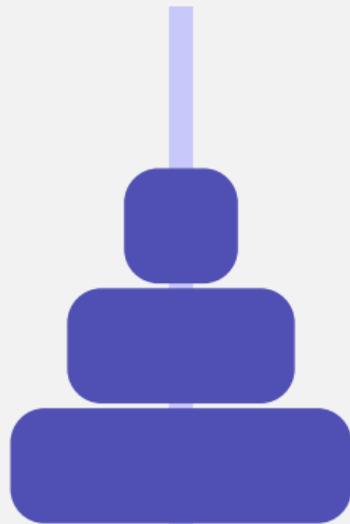


Rechts

Die Türme von Hanoi - So gehts!



Links



Mitte

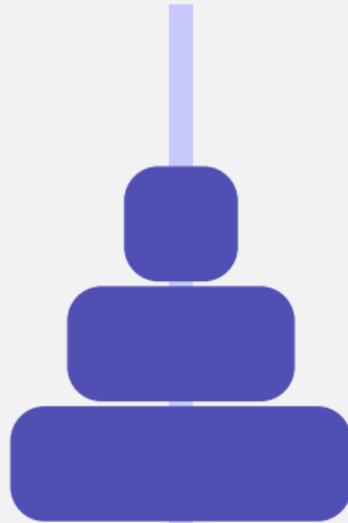


Rechts

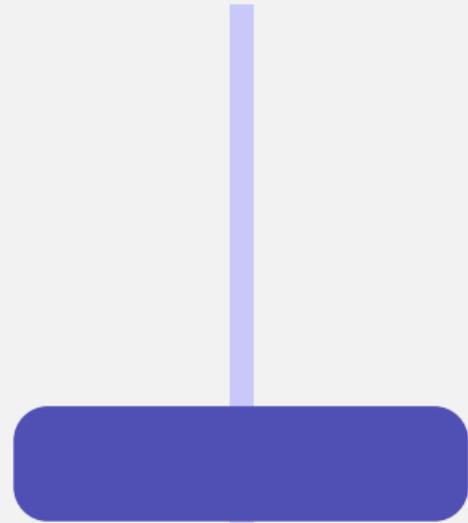
Die Türme von Hanoi - So gehts!



Links



Mitte

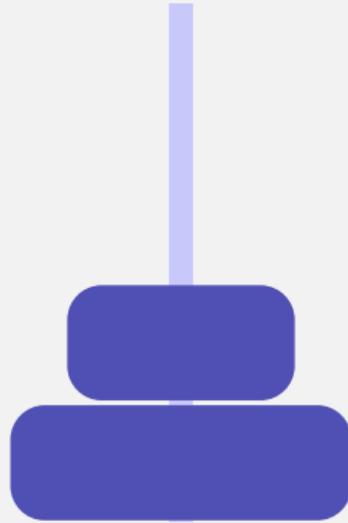


Rechts

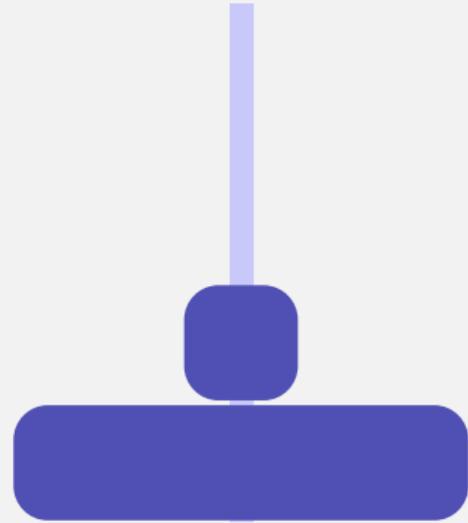
Die Türme von Hanoi - So gehts!



Links

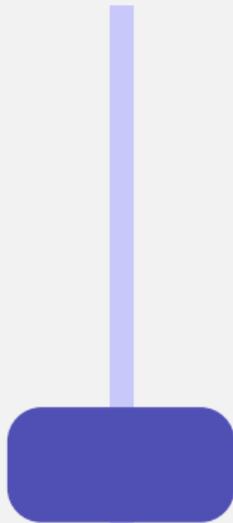


Mitte

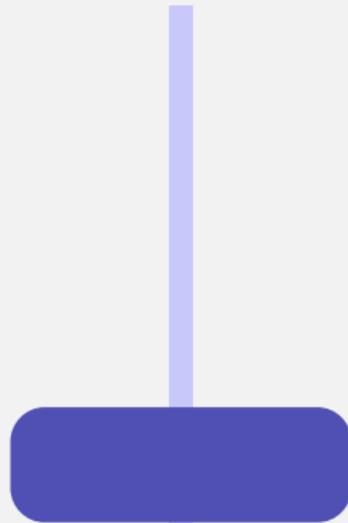


Rechts

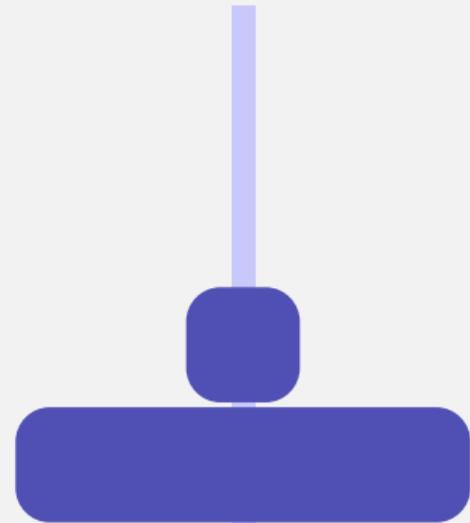
Die Türme von Hanoi - So gehts!



Links

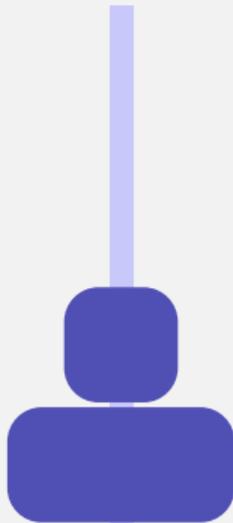


Mitte

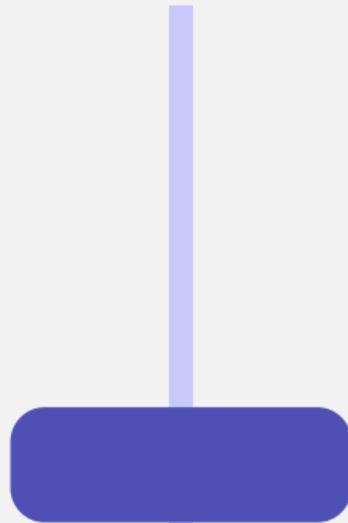


Rechts

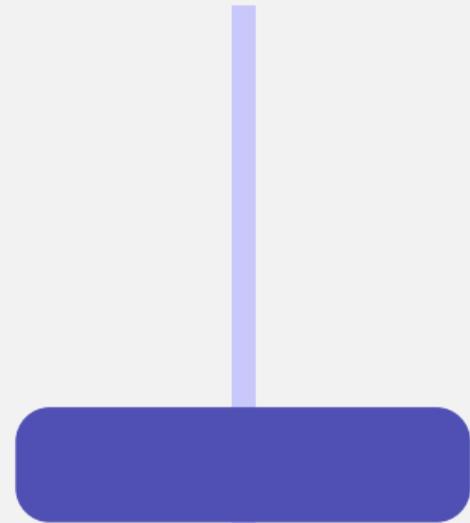
Die Türme von Hanoi - So gehts!



Links

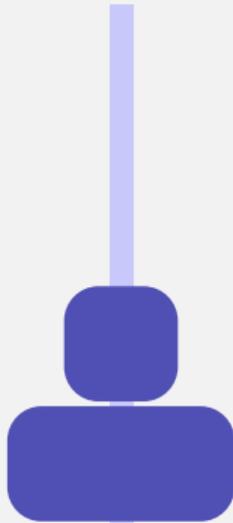


Mitte



Rechts

Die Türme von Hanoi - So gehts!



Links



Mitte



Rechts

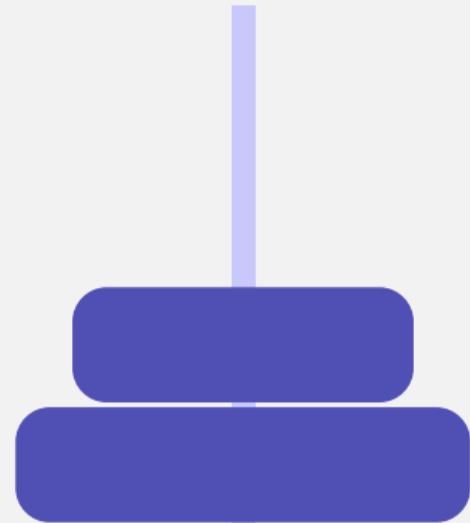
Die Türme von Hanoi - So gehts!



Links



Mitte



Rechts

Die Türme von Hanoi - So gehts!



Links



Mitte



Rechts

Die Türme von Hanoi - So gehts!



Links



Mitte



Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz



Links



Mitte



Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz



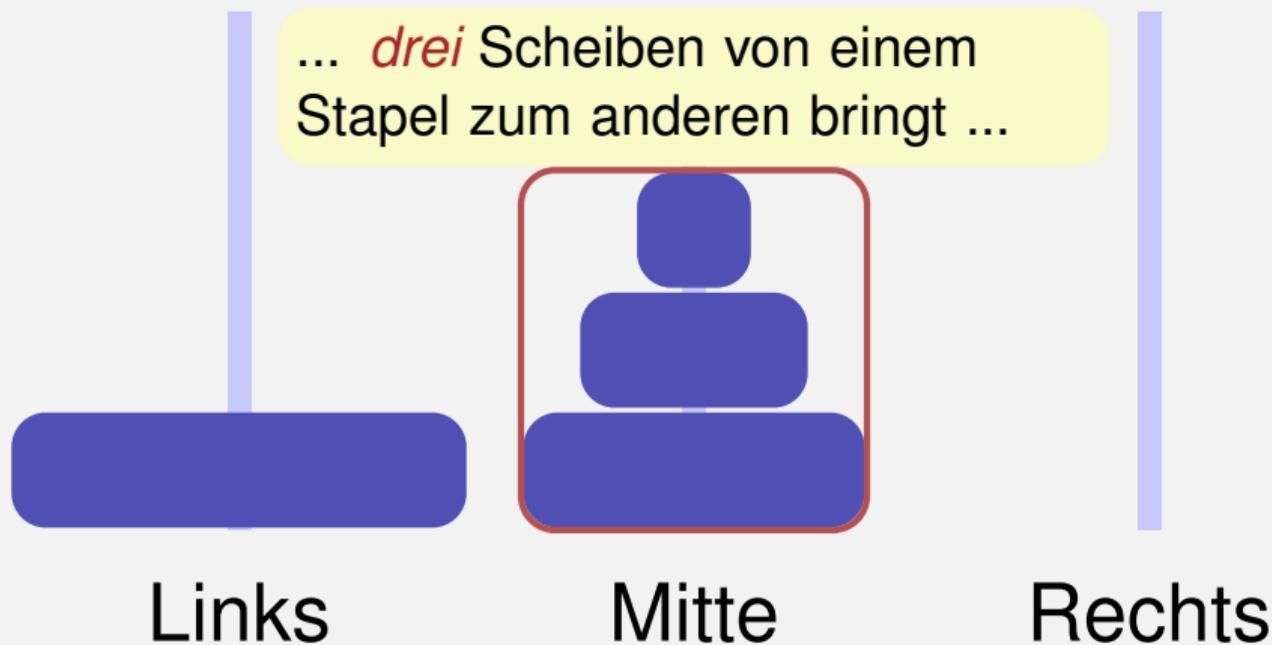
Mal *angenommen*, wir
wüssten wie man ...

Links

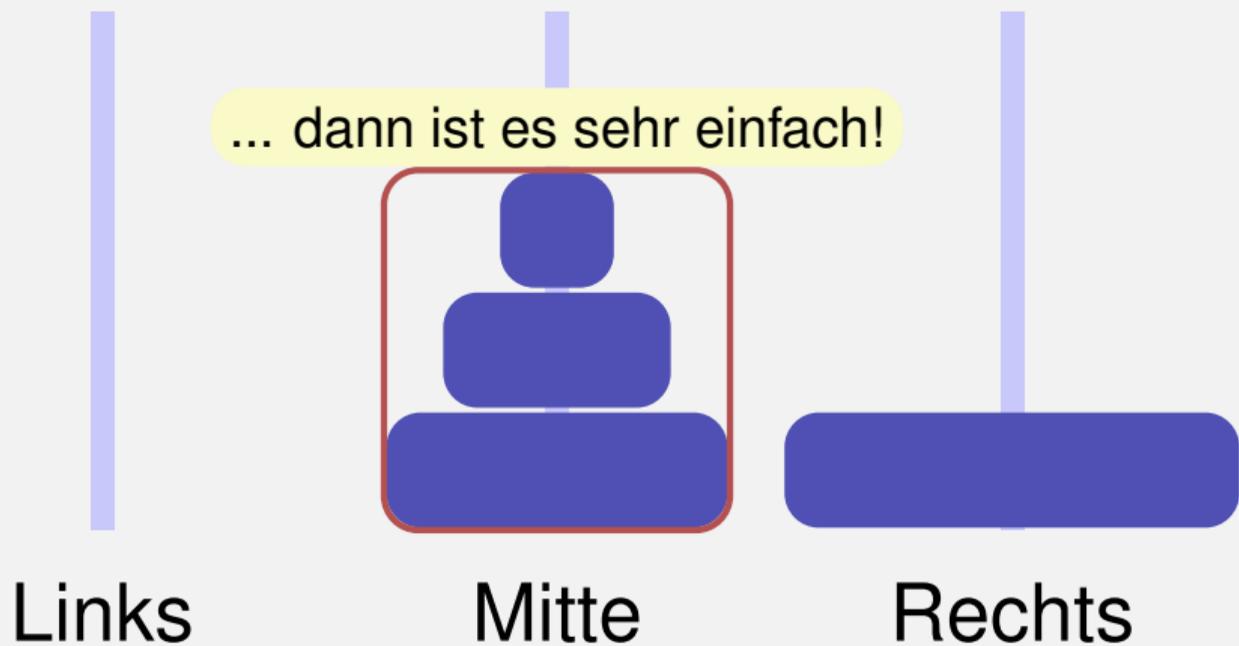
Mitte

Rechts

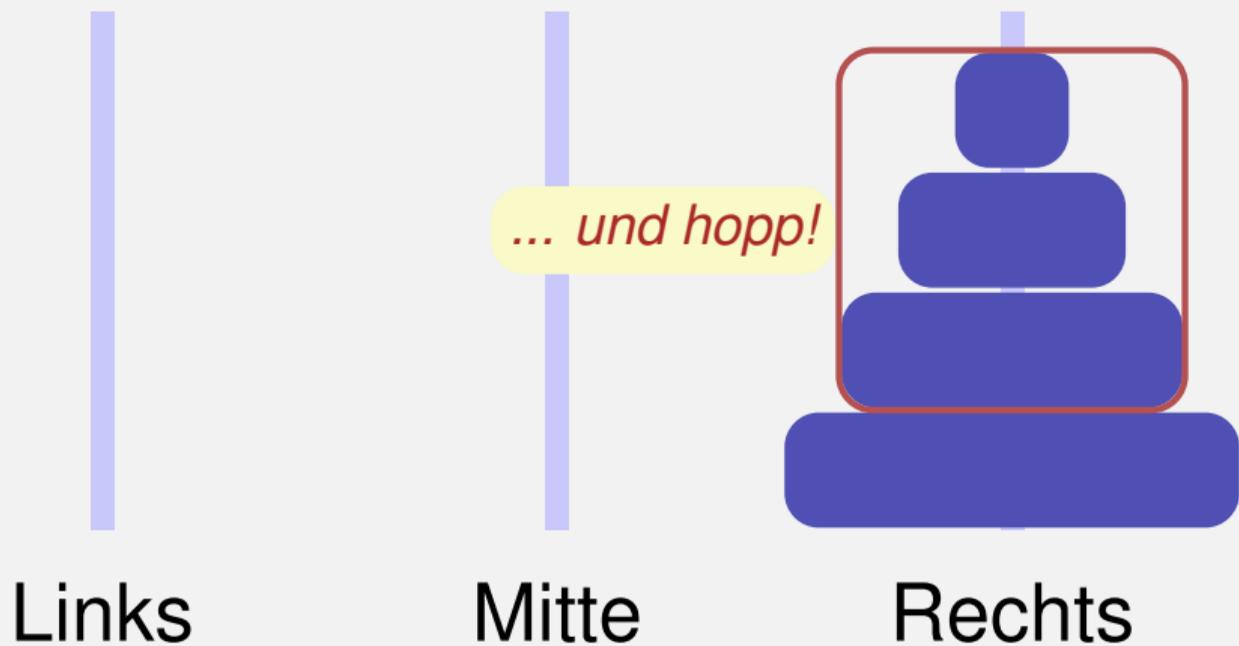
Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



Links

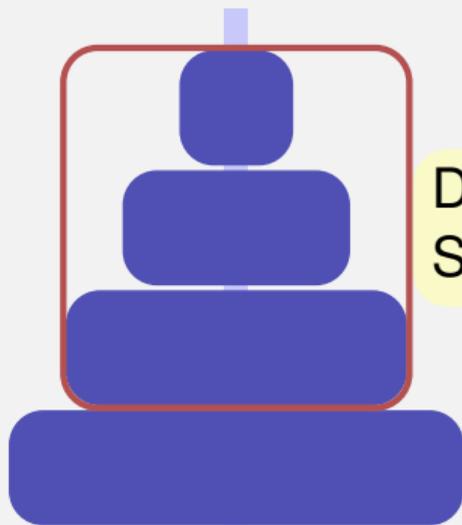


Mitte



Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz



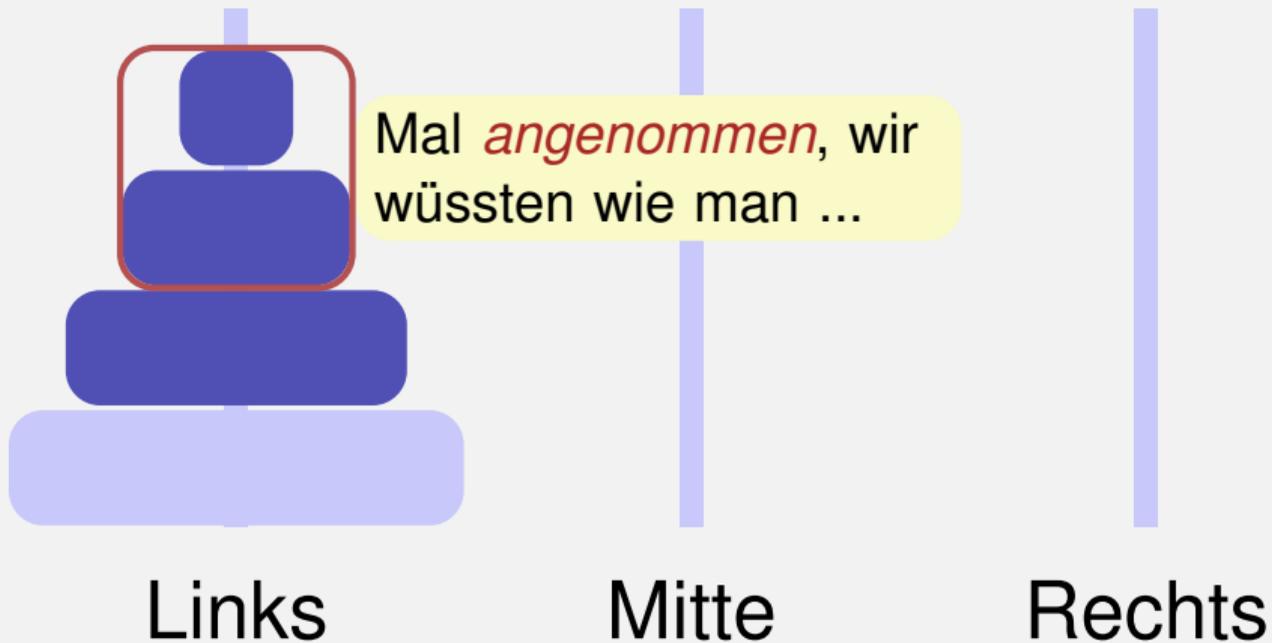
Doch *wie* können wir drei
Scheiben bewegen?

Links

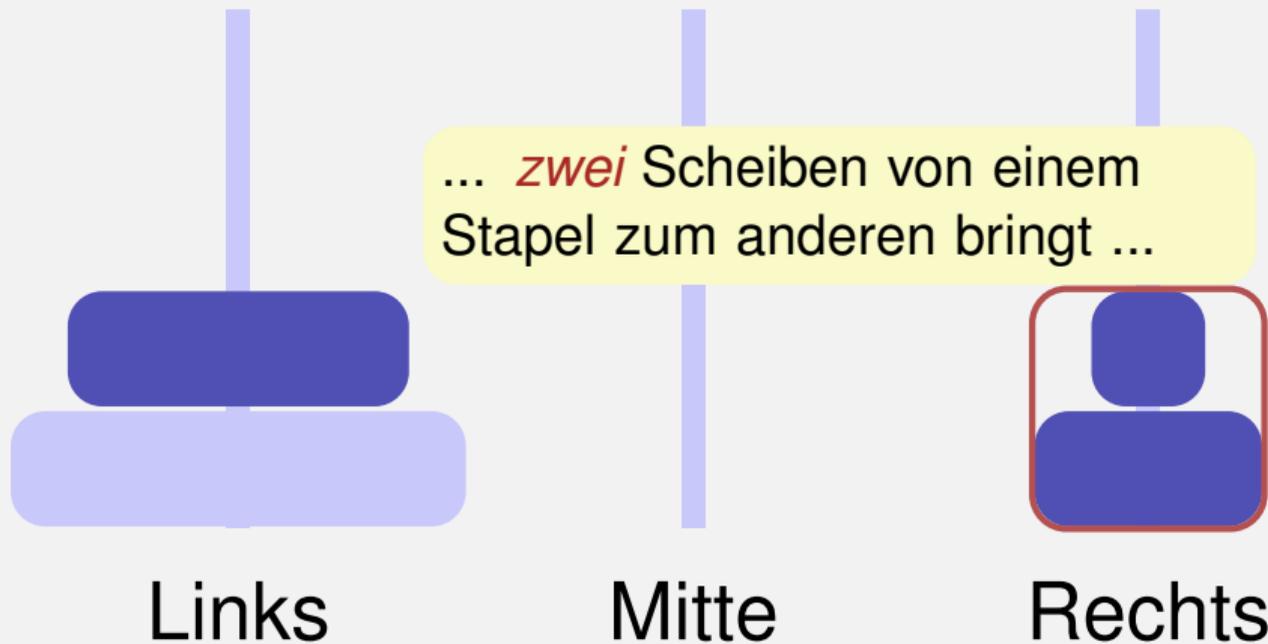
Mitte

Rechts

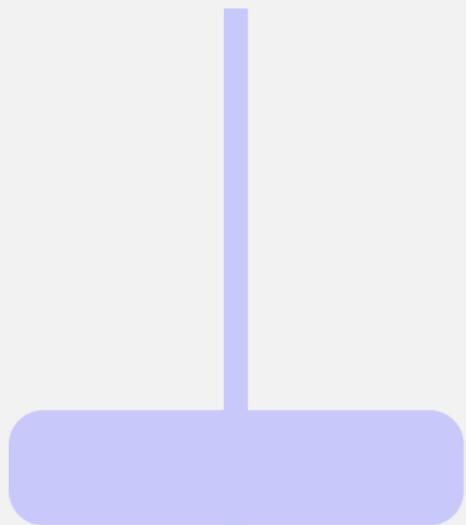
Die Türme von Hanoi - Rekursiver Lösungsansatz



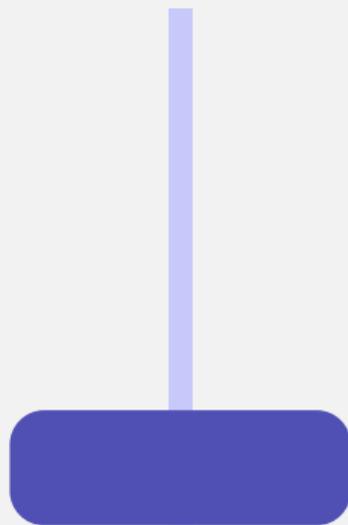
Die Türme von Hanoi - Rekursiver Lösungsansatz



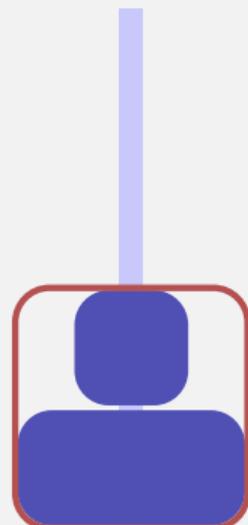
Die Türme von Hanoi - Rekursiver Lösungsansatz



Links

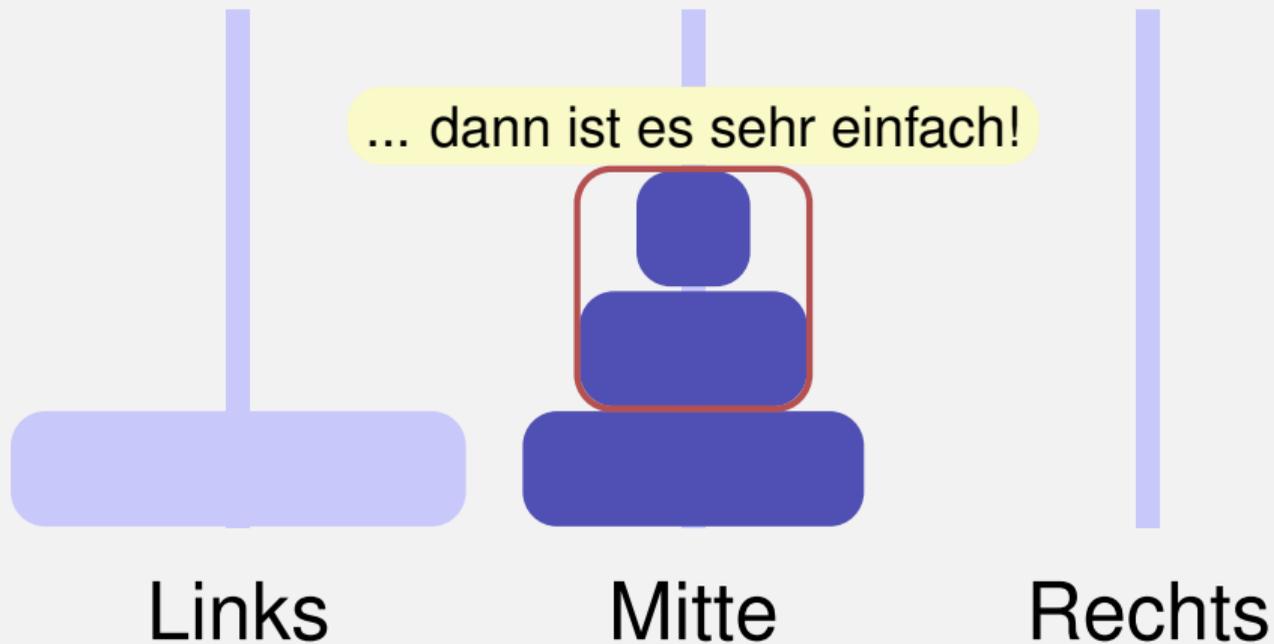


Mitte

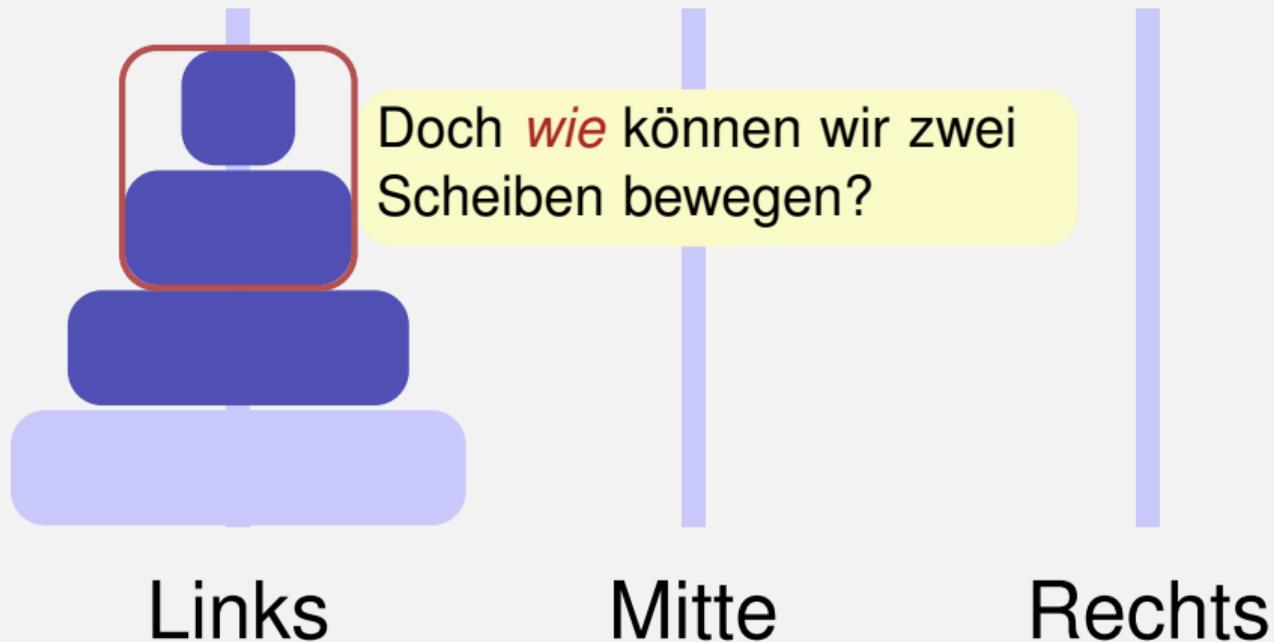


Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



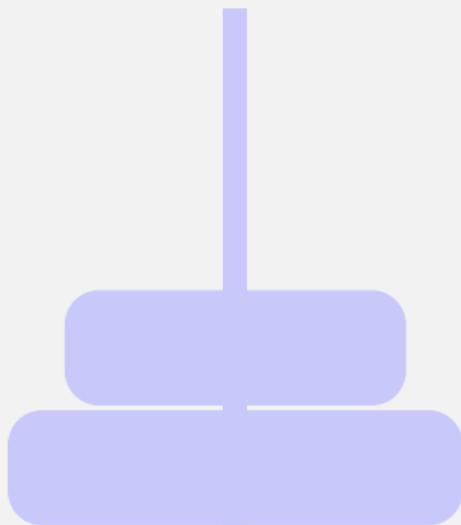
Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



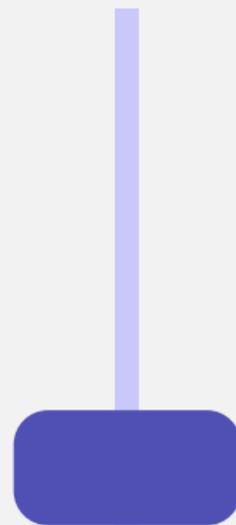
Die Türme von Hanoi - Rekursiver Lösungsansatz



Links



Mitte



Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Code



left

middle

right

Bewege 4 Scheiben von left nach right mit Hilfsstapel middle:

```
move(4, "left", "middle", "right")
```

Die Türme von Hanoi - Code

`move(n, src, aux, dst)` \Rightarrow

- 1 Bewege die obersten $n - 1$ Scheiben von *src* nach *aux* mit Hilfsstapel *dst*:
`move(n - 1, src, dst, aux);`
- 2 Bewege 1 Scheibe von *src* nach *dst*
`move(1, src, aux, dst);`
- 3 Bewege die obersten $n - 1$ Scheiben von *aux* nach *dst* mit Hilfsstapel *src*:
`move(n - 1, aux, src, dst);`

Die Türme von Hanoi - Code

```
void move(int n, const string &src, const string &aux, const string &dst){  
    if (n == 1) {  
        // base case ('move' the disc)  
        std::cout << src << " --> " << dst << std::endl;  
    } else {  
        // recursive case  
  
    }  
}
```

Die Türme von Hanoi - Code

```
void move(int n, const string &src, const string &aux, const string &dst){  
    if (n == 1) {  
        // base case ('move' the disc)  
        std::cout << src << " --> " << dst << std::endl;  
    } else {  
        // recursive case  
        move(n-1, src, dst, aux);  
  
    }  
}
```

Die Türme von Hanoi - Code

```
void move(int n, const string &src, const string &aux, const string &dst){
    if (n == 1) {
        // base case ('move' the disc)
        std::cout << src << " --> " << dst << std::endl;
    } else {
        // recursive case
        move(n-1, src, dst, aux);
        move(1, src, aux, dst);
    }
}
```

Die Türme von Hanoi - Code

```
void move(int n, const string &src, const string &aux, const string &dst){  
    if (n == 1) {  
        // base case ('move' the disc)  
        std::cout << src << " --> " << dst << std::endl;  
    } else {  
        // recursive case  
        move(n-1, src, dst, aux);  
        move(1, src, aux, dst);  
        move(n-1, aux, src, dst);  
    }  
}
```

Die Türme von Hanoi - Code

```
void move(int n, const string &src, const string &aux, const string &dst){
    if (n == 1) {
        // base case ('move' the disc)
        std::cout << src << " --> " << dst << std::endl;
    } else {
        // recursive case
        move(n-1, src, dst, aux);
        move(1, src, aux, dst);
        move(n-1, aux, src, dst);
    }
}

int main() {
    move(4, "left", "middle", "right");
    return 0;
}
```

Die Türme von Hanoi - Code Alternative

```
void move(int n, const string &src, const string &aux, const string &dst){  
    // base case  
    if (n == 0) return;  
  
    // recursive case  
    move(n-1, src, dst, aux);  
    std::cout << src << " --> " << dst << "\n";  
    move(n-1, aux, src, dst);  
}
```

```
int main() {  
    move(4, "left ", "middle", "right ");  
    return 0;  
}
```