

14. 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
```

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

Felder: Indizes vs. Zeiger



```
int a[n];
```

```
// Aufgabe: setze alle Elemente auf 0
```

```
// Lösung mit Indizes ist lesbarer
```

```
for (int i = 0; i < n; ++i)
    a[i] = 0;
```

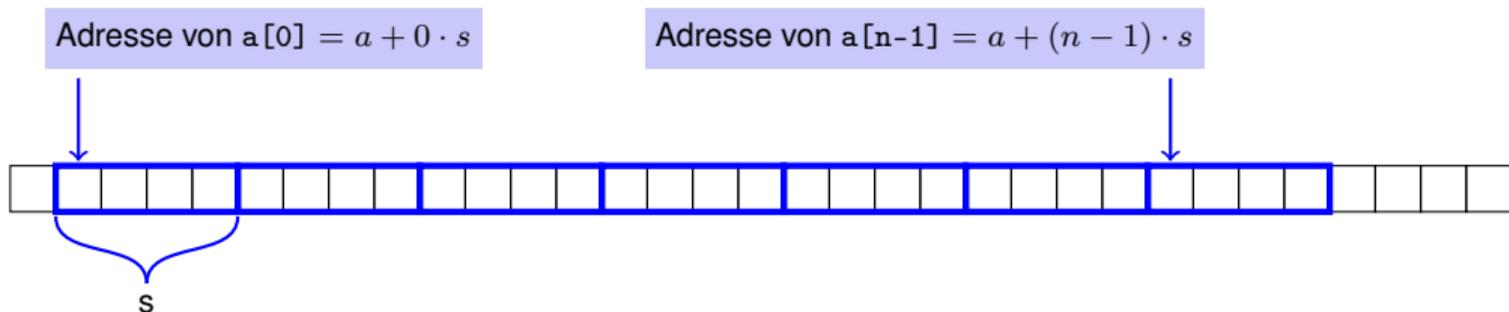
```
// Lösung mit Zeigern ist schneller und allgemeiner
```

```
int* begin = a; // Zeiger aufs erste Element
int* end = a+n; // Zeiger hinter das letzte Element
for (int* p = begin; p != end; ++p)
    *p = 0;
```

Felder und Indizes

```
// Setze alle Elemente auf value  
for (int i = 0; i < n; ++i)  
    a[i] = value;
```

Berechnungsaufwand



⇒ Eine **Addition** und eine **Multiplikation** pro Element

Die Wahrheit über wahlfreien Zugriff

Der Ausdruck

$a[i]$

ist äquivalent zu

$*(a + i)$

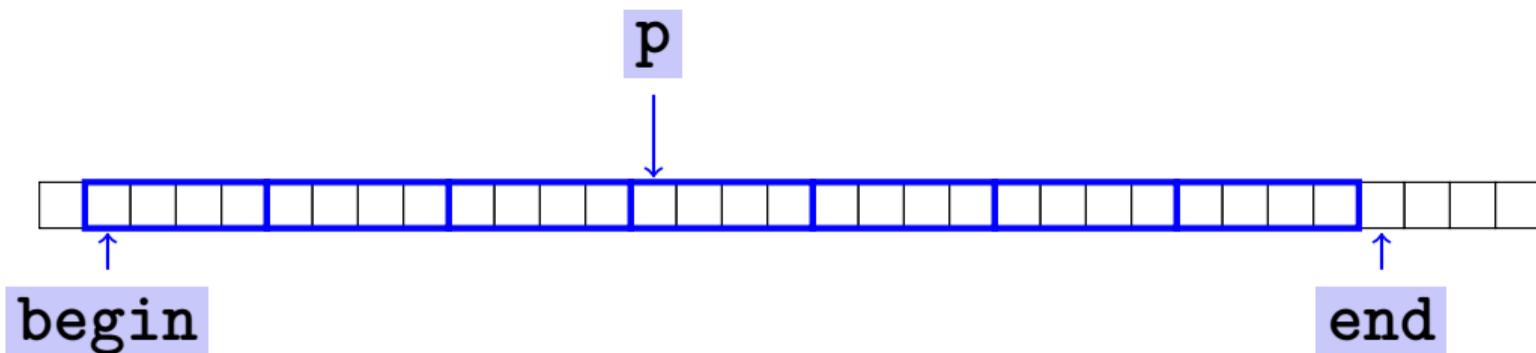


$a + i \cdot s$

Felder und Zeiger

```
// Setze alle Elemente auf value  
for (int* p = begin; p != end; ++p)  
    *p = value;
```

Berechnungsaufwand



⇒ eine **Addition** pro Element

Wahlfreier Zugriff

- öffne Buch auf S.1
- Klappe Buch zu
- öffne Buch auf S.2-3
- Klappe Buch zu
- öffne Buch auf S.4-5
- Klappe Buch zu
-

Sequentieller Zugriff

- öffne Buch auf S.1
- blättere um
- ...

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 (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)
- 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:

- `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`

```
// 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
```

Übergabe der Adresse (des ersten Elements) von a

Zeigerwerte sind keine Ganzzahlen

- Adressen können als „Hausnummern des Speichers“, also als Zahlen interpretiert werden.
- Ganzzahl- und Zeigerarithmetik verhalten sich aber unterschiedlich.

`ptr + 1` ist *nicht* die nächste Hausnummer, sondern die *s*-nächste, wobei *s* der Speicherbedarf eines Objekts des Typs ist, der `ptr` zugrundeliegt.

- Zeiger und Ganzzahlen sind nicht kompatibel:

```
int* ptr = 5; // Fehler: invalid conversion from int to int*
int a = ptr;  // Fehler: invalid conversion from int* to int
```

Null-Zeiger

- spezieller Zeigerwert, der angibt, dass noch auf kein Objekt gezeigt wird
- repräsentiert durch die ganze Zahl 0 (konvertierbar nach T*)

```
int* iptr = 0;
```

- kann nicht dereferenziert werden (prüfbar zur Laufzeit)
- zur Vermeidung undefinierten Verhaltens

```
int* iptr; // iptr zeigt ‘ins Nirvana’  
int j = *iptr; // Illegale Adresse in *
```

Zeigersubtraktion

- Wenn $p1$ und $p2$ auf Elemente desselben Arrays a mit Länge n zeigen
- und $0 \leq k_1, k_2 \leq n$ die Indizes der Elemente sind, auf die $p1$ und $p2$ zeigen, so gilt

$p1 - p2$ hat den Wert $k_1 - k_2$



Nur gültig, wenn $p1$ und $p2$ ins gleiche Feld zeigen.

- Die Zeigerdifferenz beschreibt, „wie weit die Elemente voneinander entfernt sind“

Zeigeroperatoren

Beschreibung	Op	Stelligkeit	Präzedenz	Assoziativität	Zuordnung
Subskript	[]	2	17	links	R-Werte → L-Wert
Dereferenzierung	*	1	16	rechts	R-Wert → L-Wert
Adresse	&	1	16	rechts	L-Wert → R-Wert

Präzedenzen und Assoziativitäten von +, -, ++ (etc.) wie in Kapitel 2

Mutierende Funktionen

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden.

Beispiel

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

Übergabe der Adresse des Elements hinter a

Übergabe der Adresse (des ersten Elements) von a

- Solche Funktionen heissen *mutierend*

Const-Korrektheit

- Es gibt auch *nicht* mutierende Funktionen, die nur lesend auf Elemente eines Feldes zugreifen

```
// 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;
}
```

- Kennzeichnung mit `const`: Objekte können durch solche `const`-Zeiger nicht im Wert verändert werden.

const ist nicht absolut

- Der Wert an einer Adresse kann sich ändern, auch wenn ein const-Zeiger diese Adresse speichert.

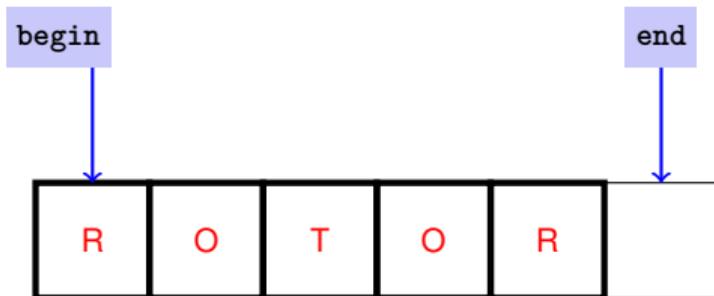
beispiel

```
int a[5];
const int* begin1 = a;
int*      begin2 = a;
*begin1 = 1;    // Fehler: *begin1 ist const
*begin2 = 1;    // ok, obwohl sich damit auch *begin1 ändert
```

- const ist ein Versprechen lediglich aus Sicht des const-Zeigers, keine absolute Garantie.

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;
}
```



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

Vorteile der Verwendung der Standardbibliothek

- Einfachere Programme
- Weniger Fehlerquellen
- Guter, schneller Code
- Code unabhängig vom Datentyp (nächste Folie)
- Es existieren natürlich auch Algorithmen für schwierigere Probleme, wie z.B. das (effiziente) Sortieren eines Feldes

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

- Templates erlauben die Angabe eines Typs als Argument
- Der Compiler deduziert den passenden Typ aus den Aufrufargumenten

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
- **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 (natürlich)
- 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...
- ...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};
```



8 Byte (Speicherzelle = 1 Byte = 8 Bit)

```
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.

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`

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

■ `std::vector<int>::iterator`

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

■ Ein Vektor-Iterator `it` ist kein Zeiger, verhält sich aber so:

- zeigt auf ein Vektor-Element und kann dereferenziert werden (`*it`)
- kennt Arithmetik und Vergleiche (`++it`, `it+2`, `it < end`,...)

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

- `v.begin()` zeigt auf das erste Element von `v`
- `v.end()` zeigt hinter das letzte Element von `v`
- 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`
- Dann hilft die Deklaration eines *Typ-Alias* mit

`typedef Typ Name;`

bestehender Typ

Name, unter dem der Typ
neu auch angesprochen
werden kann

Beispiele

```
typedef std::vector<int> int_vec;  
typedef int_vec::const_iterator Cvit;
```

Vektor-Iteratoren funktionieren wie Zeiger

```
typedef std::vector<int>::const_iterator Cvit;
```

```
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

```
typedef std::vector<int>::iterator Vit;
```

```
// 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.

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

- 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)

Fasse den Text als Menge von Buchstaben auf:

```
#include<set>
```

```
...
```

```
typedef std::set<char>::const_iterator Csit;
```

```
...
```

```
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)

Stelle fest, ob der Text ein Fragezeichen enthält und gib alle im Text enthaltenen Buchstaben aus

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? **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

- 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:

- 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

Warum Zeiger und Iteratoren?

Würde man nicht diesen Code

```
for (int i=0; i<n; ++i)
    a[i] = 0;
```

gegenüber folgendem Code bevorzugen?

```
for (int* ptr=a; ptr<a+n; ++ptr)
    *ptr = 0;
```

Vielleicht, aber um (hier noch besser!) das generische `std::fill(a, a+n, 0)`; benutzen zu können, *müssen* wir mit Zeigern arbeiten.

Warum Zeiger und Iteratoren?

Zur Verwendung der Standardbibliothek muss man also wissen:

- statisches Feld `a` ist zugleich ein Zeiger auf das erste Element von `a`
- `a+i` ist ein Zeiger auf das Element mit Index i

Verwendung der Standardbibliothek mit anderen Containern:
Zeiger \Rightarrow Iteratoren

Warum Zeiger und Iteratoren?

Beispiel: Zum Suchen des kleinsten Elementes eines Containers im Bereich `[begin, end)` verwende

```
std::min_element(begin, end)
```

- Gibt einen *Iterator* auf das kleinste Element zurück.
- Zum Auslesen des kleinsten Elementes muss man noch dereferenzieren:

```
*std::min_element(begin, end)
```

Darum Zeiger und Iteratoren

- Selbst für Nichtprogrammierer und “dumme” Nur-Benutzer der Standardbibliothek: Ausdrücke der Art `*std::min_element(begin, end)` lassen sich ohne die Kenntnis von Zeigern und Iteratoren nicht verstehen.
- Hinter den Kulissen der Standardbibliothek ist das Arbeiten mit dynamischem Speicher auf Basis von Zeigern unvermeidbar. Mehr dazu später in der Vorlesung!

15. Rekursion 1

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

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...
- ...nur noch schlechter ("verbrennt" Zeit **und** Speicher)

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

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  
}
```

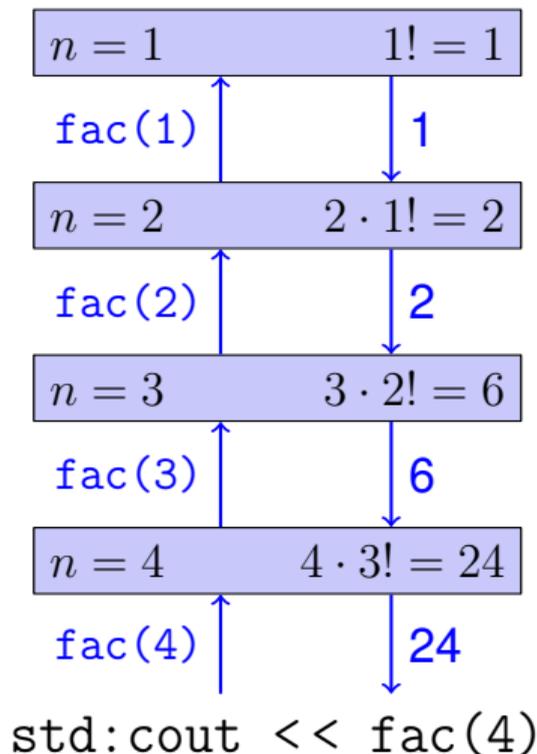
Initialisierung des formalen Arguments: $n = 4$

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

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



Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\text{gcd}(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgender mathematischen Rekursion (Beweis im Skript):

$$\text{gcd}(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \text{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

Euklidischer Algorithmus in C++

$$\text{gcd}(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \text{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
unsigned int gcd
(unsigned int a, unsigned int b)
{
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

Terminierung: $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

Fibonacci-Zahlen in C++

Laufzeit

`fib(50)` dauert „ewig“, denn es berechnet

F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal,
 F_{43} 21-mal ... F_1 ca. 10^9 mal (!)

```
unsigned int fib (unsigned int n)
```

```
{
```

```
    if (n == 0) return 0;
```

```
    if (n == 1) return 1;
```

```
    return fib (n-1) + fib (n-2); // n > 1
```

```
}
```

Korrektheit
und
Terminierung
sind klar.

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \dots, F_n!$
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen a und b)!
- Berechne die nächste Zahl als Summe von a und b!

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib (unsigned int n){  
    if (n == 0) return 0;  
    if (n <= 2) return 1;  
    unsigned int a = 1; // F_1  
    unsigned int b = 1; // F_2  
    for (unsigned int i = 3; i <= n; ++i)  
    {  
        unsigned int a_old = a; // F_{i-2}  
        a = b; // F_{i-1}  
        b += a_old; // F_{i-1} += F_{i-2} -> F_i  
    }  
    return b;  
}
```

sehr schnell auch bei fib(50)

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

a

b

Rekursion und Iteration

Rekursion kann *immer* simuliert werden durch

- Iteration (Schleifen)
- expliziten „Aufrufstapel“ (z.B. Feld).

Oft sind rekursive Formulierungen einfacher, aber manchmal auch weniger effizient.