

# 1. Zeiger, Algorithmen, Iteratoren und Container

Felder als Funktionsargumente, Zeiger, Iteratoren  
auf Vektoren, Container

# Arrays und Funktionen - Ein Wunder?

```
// Program: fill.cpp
// define and use a function to fill an array
#include<iostream>

// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}

int main()
{
    int a[5];
    fill_n (a, 5, 1);
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " ";    // 1 1 1 1 1
    std::cout << "\n";

    return 0;
}
```

# Arrays und Funktionen - Ein Wunder?

```
// Program: fill.cpp
// define and use a function to fill an array
#include<iostream>

// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}

int main()
{
    int a[5];
    fill_n (a, 5, 1);
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " ";    // 1 1 1 1 1
    std::cout << "\n";

    return 0;
}
```

Initialisierung (formales Argument) geht doch für Felder gar nicht!

# Arrays und Funktionen - Ein Wunder?

```
// Program: fill.cpp
// define and use a function to fill an array
#include<iostream>

// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}

int main()
{
    int a[5];
    fill_n (a, 5, 1);
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " ";    // 1 1 1 1 1
    std::cout << "\n";

    return 0;
}
```

Initialisierung (formales Argument) geht doch für Felder gar nicht!  
Woher weiss der Compiler, wieviele Elemente **int a[]** erwartet?

# Arrays und Funktionen - Ein Wunder?

```
// Program: fill.cpp
// define and use a function to fill an array
#include<iostream>

// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}

int main()
{
    int a[5];
    fill_n (a, 5, 1);
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " ";    // 1 1 1 1 1
    std::cout << "\n";

    return 0;
}
```

Initialisierung (formales Argument) geht doch für Felder gar nicht!  
Woher weiss der Compiler, wieviele Elemente **int a[]** erwartet?  
Das Feld **a** kommt doch als R-Wert, wie können wir es verändern?

# Arrays und Funktionen - Ein Wunder?

```
// Program: fill.cpp
// define and use a function to fill an array
#include<iostream>

// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}

int main()
{
    int a[5];
    fill_n (a, 5, 1);
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " ";    // 1 1 1 1 1
    std::cout << "\n";

    return 0;
}
```

int a[] äquivalent zu int\* a

# Arrays und Funktionen - Ein Wunder?

```
// Program: fill.cpp
// define and use a function to fill an array
#include<iostream>

// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}

int main()
{
    int a[5];
    fill_n (a, 5, 1);
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " ";    // 1 1 1 1 1
    std::cout << "\n";

    return 0;
}
```

int a[] äquivalent zu int\* a

Übergabe der Adresse von a

# Zeiger Typen

**T\***

Zeiger-Typ zum zugrunde liegenden Typ **T**.

Ein Ausdruck vom Typ **T\*** heisst **Zeiger** (auf **T**).

Wert eines Zeigers auf **T** ist die Adresse eines Objektes vom Typ **T**.

## Beispiele

`int* p;` Variable **p** ist Zeiger auf ein **int**.

`float* q;` Variable **q** ist Zeiger auf ein **float**.

# Zeiger Typen

**T\***

Zeiger-Typ zum zugrunde liegenden Typ **T**.

Ein Ausdruck vom Typ **T\*** heisst **Zeiger** (auf **T**).

Wert eines Zeigers auf **T** ist die Adresse eines Objektes vom Typ **T**.

```
int* p = ...;
```

*p: Adresse (der ersten Speicherzelle) eines int*



# Zeiger-Initialisierung

- Zeiger können (nur) mit kompatiblen Zeigerwerten (Adressen) initialisiert werden

# Zeiger-Initialisierung

- Zeiger können (nur) mit kompatiblen Zeigerwerten (Adressen) initialisiert werden

## Beispiele

```
int* i = p;      // ok: i repräsentiert nun dieselbe Adresse wie p
double* j = i;  // Fehler: cannot convert
```

# Zeiger-Initialisierung

- Zeiger können (nur) mit kompatiblen Zeigerwerten (Adressen) initialisiert werden

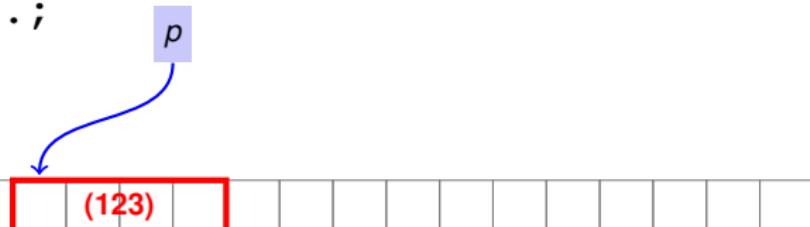
## Beispiele

```
int* i = p;      // ok: i repräsentiert nun dieselbe Adresse wie p
double* j = i;  // Fehler: cannot convert
```

- Zeiger können verglichen werden. Vergleich liefert als Ergebnis, ob die Zeigerwerte (d.h. die repräsentierten Adressen) übereinstimmen.

```
int* p = ...;
```

p



# Zeiger-Initialisierung

- Zeiger können (nur) mit kompatiblen Zeigerwerten (Adressen) initialisiert werden

## Beispiele

```
int* i = p;      // ok: i repräsentiert nun dieselbe Adresse wie p
double* j = i;  // Fehler: cannot convert
```

- Zeiger können verglichen werden. Vergleich liefert als Ergebnis, ob die Zeigerwerte (d.h. die repräsentierten Adressen) übereinstimmen.

```
int* p = ...; i
int* i = p;      p
```



# Zeiger-Initialisierung

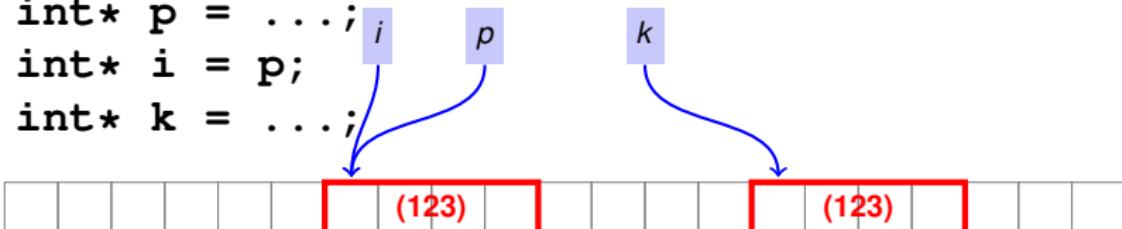
- Zeiger können (nur) mit kompatiblen Zeigerwerten (Adressen) initialisiert werden

## Beispiele

```
int* i = p;      // ok: i repräsentiert nun dieselbe Adresse wie p
double* j = i;  // Fehler: cannot convert
```

- Zeiger können verglichen werden. Vergleich liefert als Ergebnis, ob die Zeigerwerte (d.h. die repräsentierten Adressen) übereinstimmen.

```
int* p = ...; i
int* i = p;
int* k = ...;
```



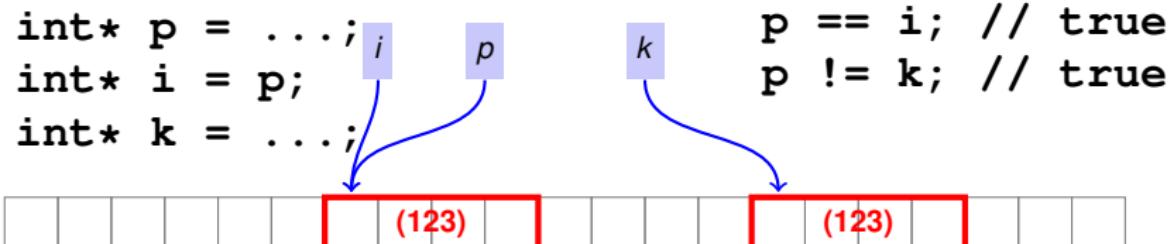
# Zeiger-Initialisierung

- Zeiger können (nur) mit kompatiblen Zeigerwerten (Adressen) initialisiert werden

## Beispiele

```
int* i = p;      // ok: i repräsentiert nun dieselbe Adresse wie p
double* j = i;  // Fehler: cannot convert
```

- Zeiger können verglichen werden. Vergleich liefert als Ergebnis, ob die Zeigerwerte (d.h. die repräsentierten Adressen) übereinstimmen.



# Konversion Feld $\Rightarrow$ Zeiger

- Statisches Feld vom Typ  $T[n]$  ist konvertierbar nach  $T^*$

# Konversion Feld $\Rightarrow$ Zeiger

- Statisches Feld vom Typ  $T[n]$  ist konvertierbar nach  $T^*$

## Beispiel

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

# Konversion Feld $\Rightarrow$ Zeiger

- Statisches Feld vom Typ  $T[n]$  ist konvertierbar nach  $T^*$

## Beispiel

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

- Längeninformation geht verloren („Felder sind primitiv“).

# Konversion Feld $\Rightarrow$ Zeiger

- Statisches Feld vom Typ  $T[n]$  ist konvertierbar nach  $T^*$

## Beispiel

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

- Längeninformation geht verloren („Felder sind primitiv“).

## Beispiel

```
void fill_n (int a[], const int n, const int value)
```

# Konversion Feld $\Rightarrow$ Zeiger

- Statisches Feld vom Typ  $T[n]$  ist konvertierbar nach  $T^*$

## Beispiel

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

- Längeninformation geht verloren („Felder sind primitiv“).

## Beispiel

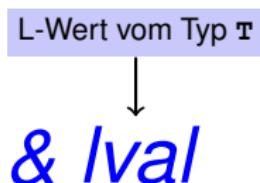
```
void fill_n (int a[], const int n, const int value)
```

ist äquivalent zu

```
void fill_n (int* a, const int n, const int value)
```

# Adress-Operator

Der Ausdruck



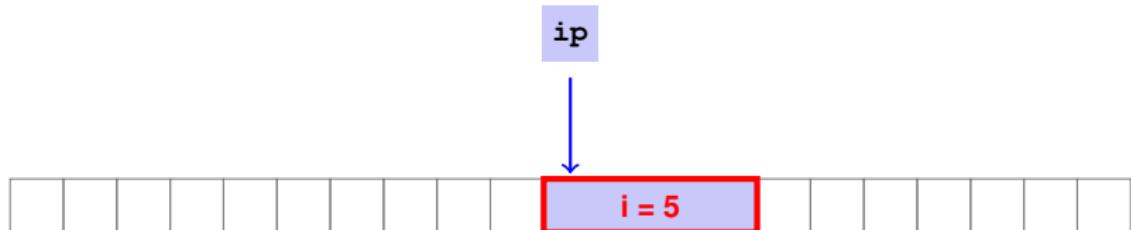
liefert als R-Wert einen *Zeiger* vom Typ  $T^*$  auf das Objekt an der Adresse von  $lval$

Der Operator  $\&$  heisst **Adress-Operator**.

# Adress-Operator

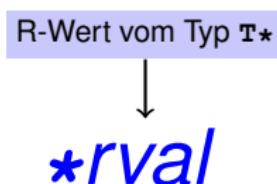
## Beispiel

```
int i = 5;  
int* ip = &i; // ip initialisiert  
              // mit Adresse von i.
```



# Dereferenz-Operator

Der Ausdruck



liefert als L-Wert den *Wert* des Objekts an der durch  $rval$  repräsentierten Adresse

Der Operator  $*$  heisst **Dereferenz-Operator**.

# Dereferenz-Operator

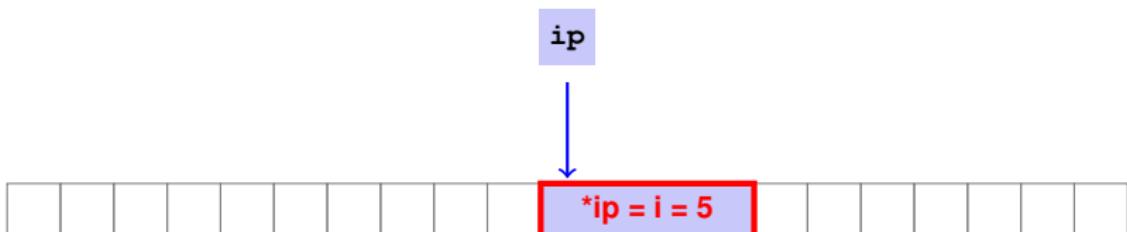
## Beispiel

```
int i = 5;
int* ip = &i;    // ip initialisiert
                  // mit Adresse von i.
int j = *ip;    // j == 5
```

# Derefenz-Operator

## Beispiel

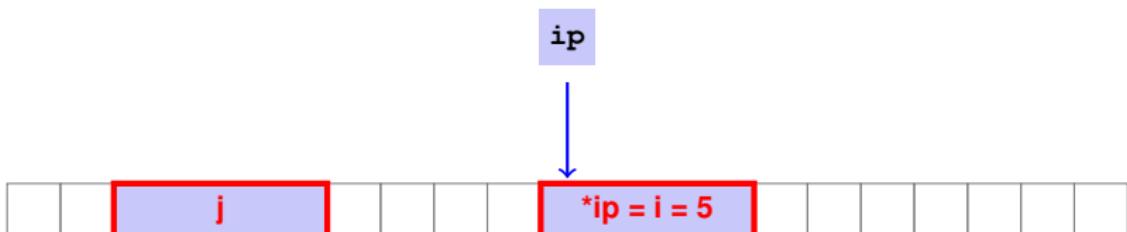
```
int i = 5;  
int* ip = &i; // ip initialisiert  
              // mit Adresse von i.  
int j = *ip; // j == 5
```



# Derefenz-Operator

## Beispiel

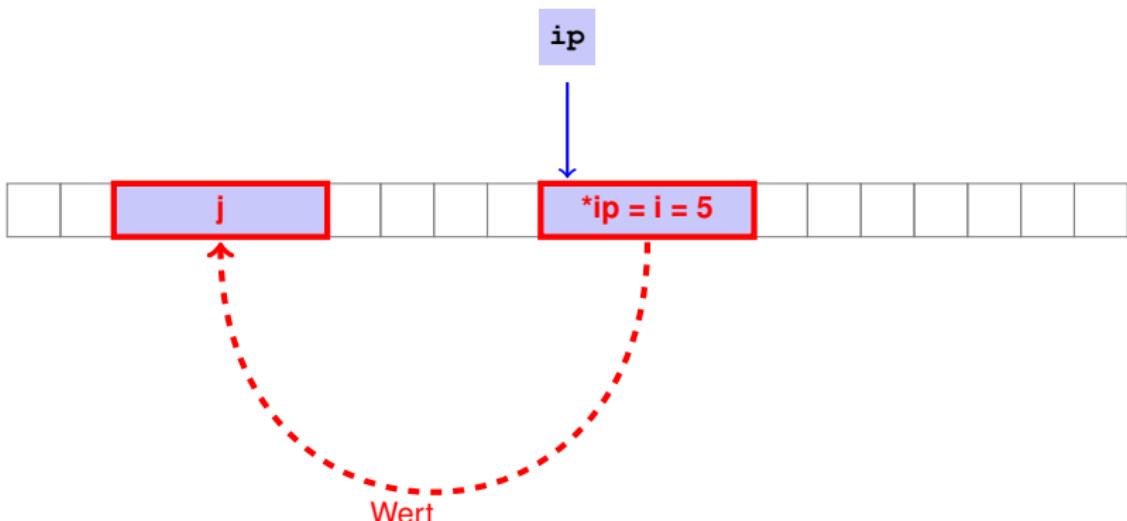
```
int i = 5;  
int* ip = &i; // ip initialisiert  
              // mit Adresse von i.  
int j = *ip; // j == 5
```



# Derefenz-Operator

## Beispiel

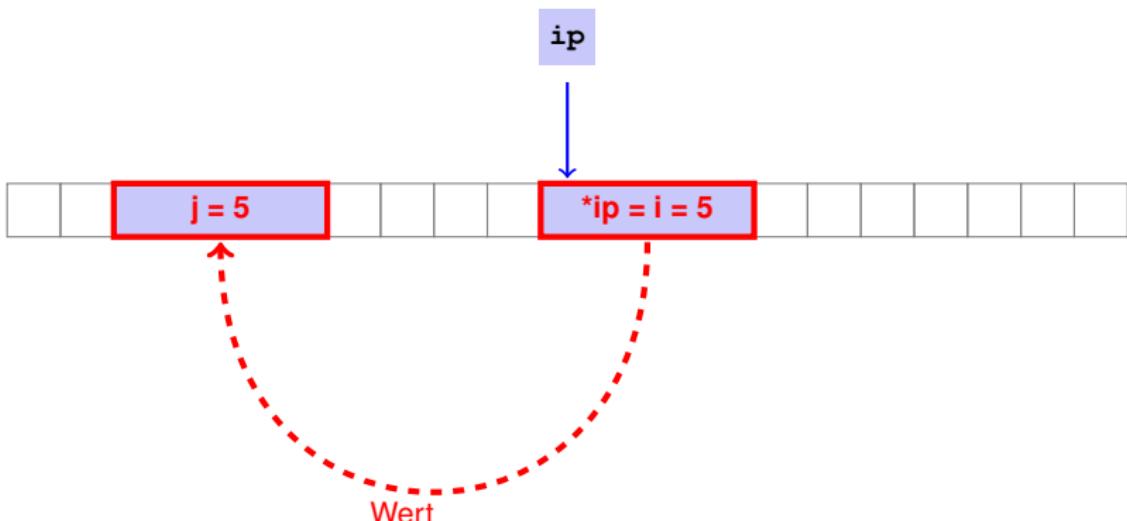
```
int i = 5;  
int* ip = &i; // ip initialisiert  
              // mit Adresse von i.  
int j = *ip; // j == 5
```



# Derefenz-Operator

## Beispiel

```
int i = 5;  
int* ip = &i; // ip initialisiert  
              // mit Adresse von i.  
int j = *ip; // j == 5
```



# Eselsbrücke

## Die Deklaration

**T\* p;**      p ist vom Typ “Zeiger auf T”

# Eselsbrücke

Die Deklaration

**T\* p;**      p ist vom Typ “Zeiger auf T”

kann gelesen werden als

**T \*p;**      \*p ist vom Typ T

# Eselsbrücke

## Die Deklaration

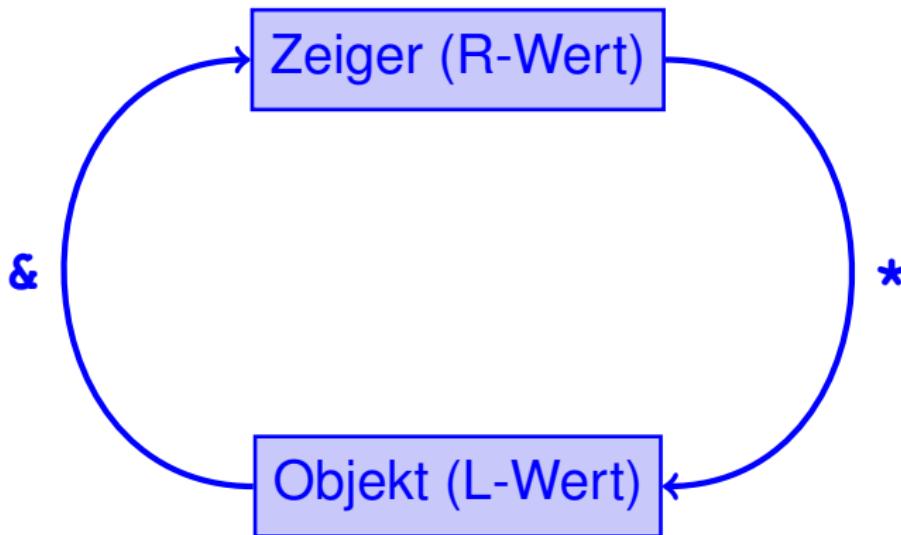
**T\* p;**      p ist vom Typ “Zeiger auf T”

kann gelesen werden als

**T \*p;**      **\*p** ist vom Typ T

Obwohl das legal ist, schreiben  
wir es nicht so!

# Adress- und Dereferenzoperator



# 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 *
```

# Zeiger-Arithmetik

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

# Zeiger-Arithmetik

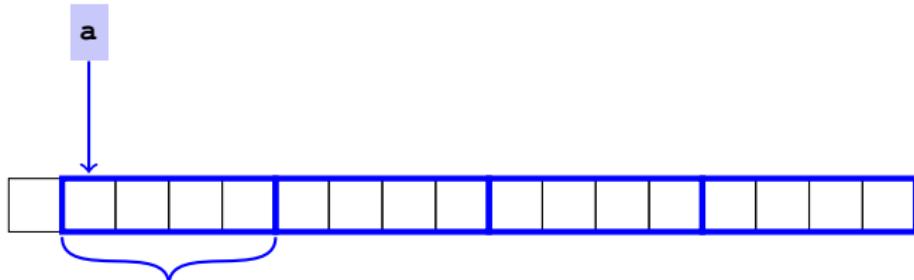
```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int* a, const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

Was heisst das bei einem Zeiger?

# Zeiger-Arithmetik

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int* a, const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

Was heisst das bei einem Zeiger?

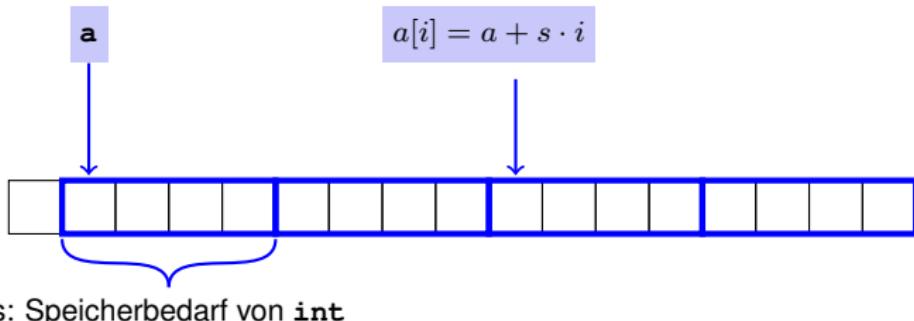


s: Speicherbedarf von int

# Zeiger-Arithmetik

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int* a, const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

Was heisst das bei einem Zeiger?



# Zeiger-Arithmetik: Zeiger plus int

- Wenn  $ptr$  ein Zeiger auf das Element mit Index  $k$  in einem Array  $a$  der Länge  $n$  ist



# Zeiger-Arithmetik: Zeiger plus int

- Wenn  $ptr$  ein Zeiger auf das Element mit Index  $k$  in einem Array  $a$  der Länge  $n$  ist
- und der Wert von  $expr$  eine ganze Zahl  $i$  ist,  
 $0 \leq k + i \leq n$ ,



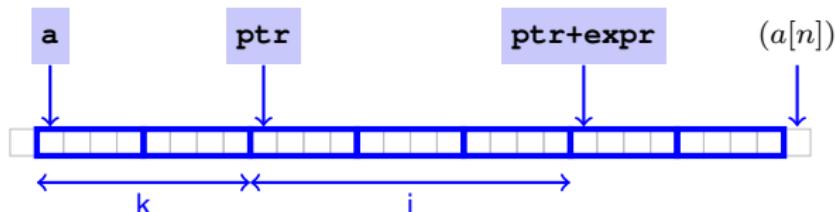
# Zeiger-Arithmetik: Zeiger plus int

- Wenn  $ptr$  ein Zeiger auf das Element mit Index  $k$  in einem Array  $a$  der Länge  $n$  ist
- und der Wert von  $expr$  eine ganze Zahl  $i$  ist,  
 $0 \leq k + i \leq n$ ,

dann liefert der Ausdruck

$ptr + expr$

einen Zeiger zum Element von  $a$  mit Index  $k + i$ .



# Zeiger-Arithmetik: Zeiger plus int

- Wenn  $ptr$  ein Zeiger auf das Element mit Index  $k$  in einem Array  $a$  der Länge  $n$  ist
- und der Wert von  $expr$  eine ganze Zahl  $i$  ist,  
 $0 \leq k + i \leq n$ ,

dann liefert der Ausdruck

$ptr + expr$

einen Zeiger zum Element von  $a$  mit Index  $k + i$ .

Für  $k + i = n$  erhalten wir einen *past-the-end*-Zeiger, der nicht dereferenziert werden darf.

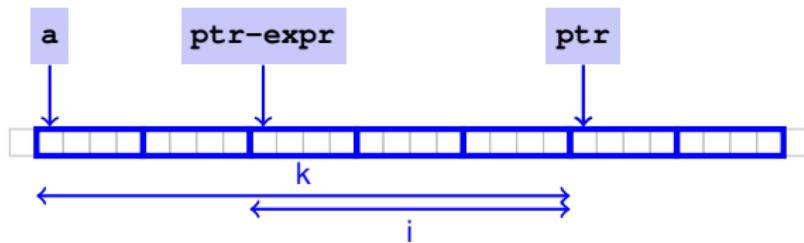
# Zeiger-Arithmetik: Zeiger minus int

- Wenn  $ptr$  ein Zeiger auf das Element mit Index  $k$  in einem Array  $a$  der Länge  $n$  ist
- und der Wert von  $expr$  eine ganze Zahl  $i$  ist,  
 $0 \leq k - i \leq n$ ,

dann liefert der Ausdruck

$ptr - expr$

einen Zeiger zum Element von  $a$  mit Index  $k - i$ .



# Die Wahrheit über wahlfreien Zugriff

Der Ausdruck

Zeiger                    int  
↓                        ↓  
*ptr[expr]*

ist äquivalent zu

$*(ptr + expr)$

# Die Wahrheit über wahlfreien Zugriff

Der Ausdruck

Zeiger  
↓  
*ptr[expr]*  
int  
↓

ist äquivalent zu

$*(ptr + expr)$

## Beispiel

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

äquivalent zu

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

# Zeigerwerte sind keine Ganzzahlen

- Adressen können als „Hausnummern des Speichers“, also als Zahlen interpretiert werden.

# Zeigerwerte sind keine Ganzzahlen

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

# 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.

# 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
```

# Zeigervergleiche

- Neben `==` und `!=` lassen sich auch `<`, `<=`, `>` und `>=` zum Vergleich von Zeigern verwenden
- Resultat des Vergleichs von Zeigern  $p1$  und  $p2$  nur spezifiziert, wenn  $p1$  und  $p2$  auf Elemente desselben Arrays  $\mathbf{a}$  zeigen.
- Ist  $n$  die Länge von  $\mathbf{a}$  und sind  $0 \leq k_1, k_2 \leq n$  die Indizes der Elemente, auf die  $p1$  und  $p2$  zeigen, so gilt

$$p1 \ (op) \ p2 \qquad \iff \qquad k_1 \ (op) \ k_2$$

# Beispiel Zeigervergleiche

```
// PRE: first zeigt auf den ersten Buchstaben, last auf den
//       letzten Buchstaben eines Feldes von Zeichen
// POST: gibt genau dann true zurueck, wenn der Bereich
//       von Zeichen ein Palindrom bildet
bool is_palindrome (const char* first, const char* last) {
    while (first < last) {
        if (*first != *last) return false;
        ++first; // eins vor
        --last; // eins zurueck
    }
    return true;
}
```

# Beispiel Zeigervergleiche

```
// PRE: first zeigt auf den ersten Buchstaben, last auf den
//       letzten Buchstaben eines Feldes von Zeichen
// POST: gibt genau dann true zurück, wenn der Bereich
//       von Zeichen ein Palindrom bildet
bool is_palindrome (const char* first, const char* last) {
    while (first < last) {
        if (*first != *last) return false;
        ++first; // eins vor
        --last; // eins zurück
    }
    return true;
}
```

## Vorteil dieser Lösung

- Adressberechnung relativ zum Feldursprung entfällt

# 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

# 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

$p1 - p2$  ist äquivalent zu  $k_1 - k_2$



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

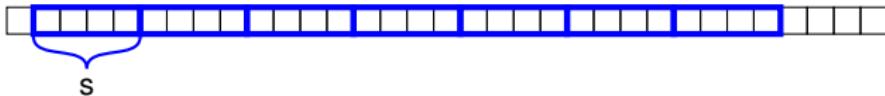
Präzedenzen und Assoziativitäten von `+`, `-`, `++` (etc.) wie in Kapitel 2

# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

```
i=0;
```



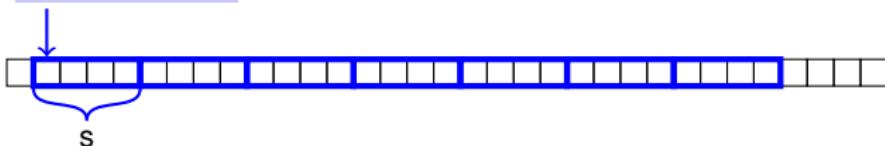
# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$i=0;$

$$a[0] = a + 0 \cdot s$$

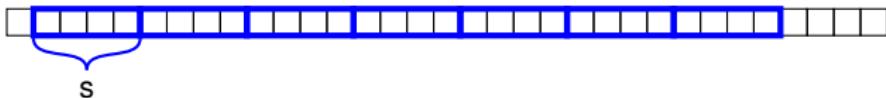


# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i$ ; ( $i = 1$ )



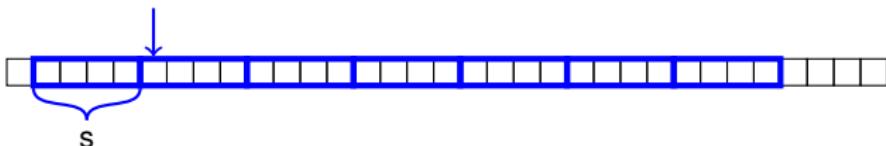
# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i; (i = 1)$

$$a[1] = a + 1 \cdot s$$

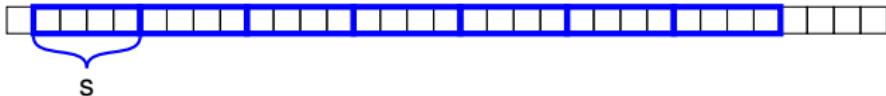


# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i$ ; ( $i = 2$ )



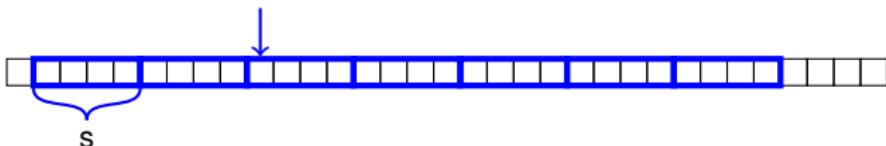
# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i; (i = 2)$

$$a[2] = a + 2 \cdot s$$

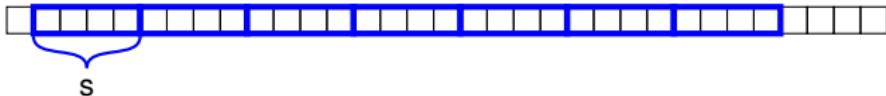


# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i$ ; ( $i = 3$ )



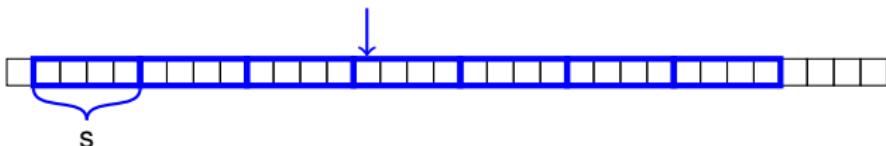
# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i; (i = 3)$

$$a[3] = a + 3 \cdot s$$

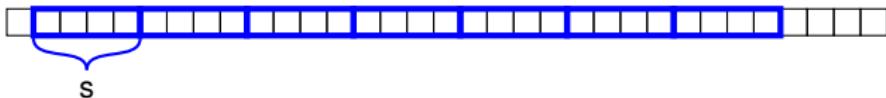


# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i$ ; ( $i = n - 1$ )



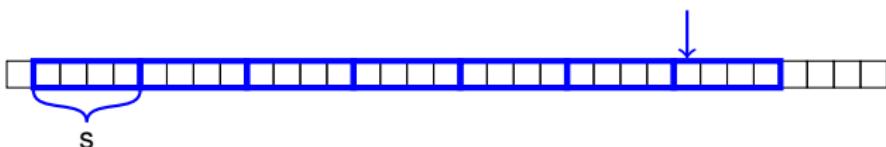
# Traversieren von Feldern – Wahlfreier Zugriff

```
// PRE: a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], const int n, const int value) {
    for (int i=0; i<n; ++i)
        a[i] = value;
}
```

## Berechnungsaufwand

$++i; (i = n - 1)$

$$a[n - 1] = a + (n - 1) \cdot s$$



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

# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

Neu: Übergabe des zu füllenden Bereichs mit Hilfe zweier Zeiger:

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

# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

Neu: Übergabe des zu füllenden Bereichs mit Hilfe zweier Zeiger:

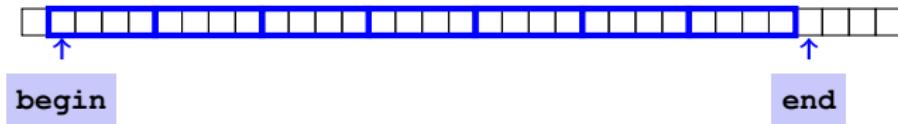
- **begin**: Zeiger auf das erste Element
- **end**: Zeiger *hinter* das letzte Element
- Das halboffene Intervall **[begin, end)** bezeichnet diesen Zeigerbereich.
- Gültiger Bereich heisst: unter diesen Adressen “leben” wirklich Feldelemente.

# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

```
ptr=begin;
```

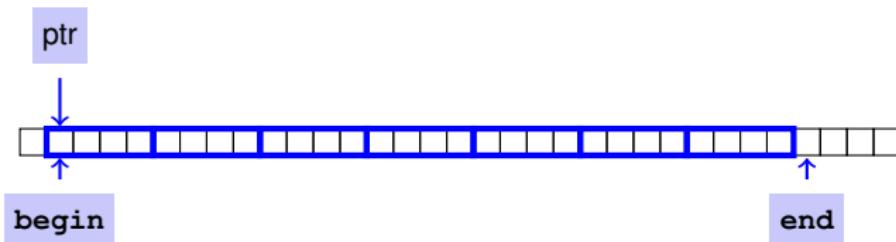


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

```
ptr=begin;
```

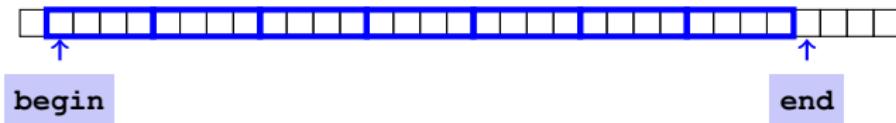


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`

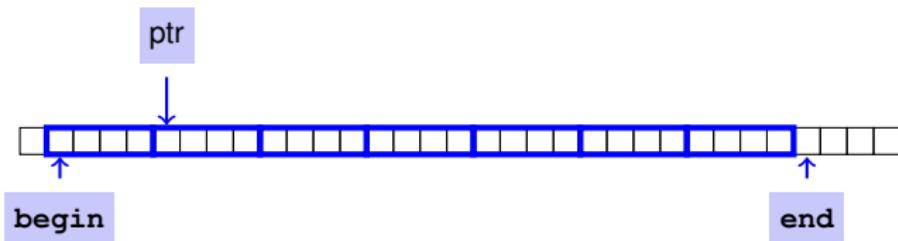


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`

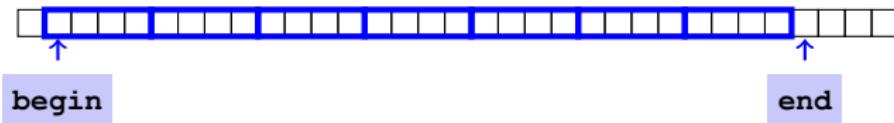


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`

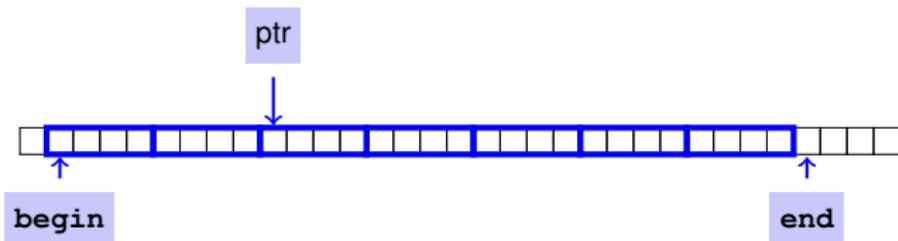


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`



# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`

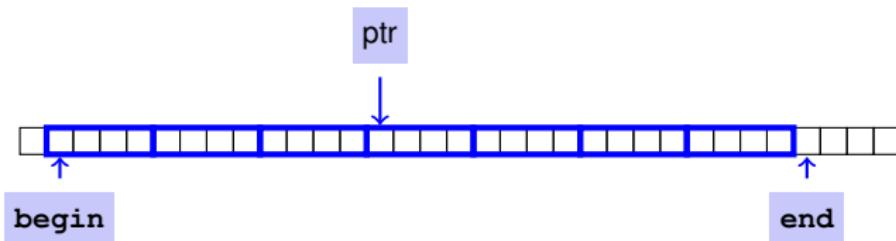


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`

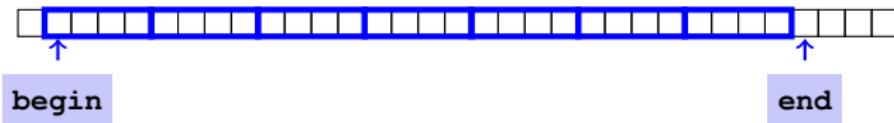


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`

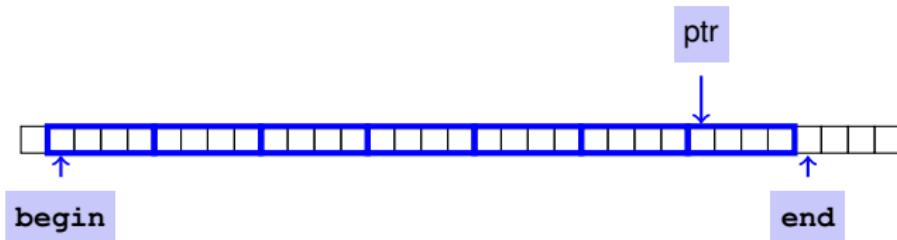


# Traversieren von Feldern – Natürliche Iteration

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
void fill (int* begin , int* end, const int value) {
    for (int* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}
```

## Berechnungsaufwand

`++ptr;`



⇒ eine **Addition** pro Feldelement

# Buchlesen: Wahlfreier Zugriff vs. natürliche Iteration

## 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
- ....

## Natürliche Iteration

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

# Mutierende Funktionen

- Zeigerargumente können verwendet werden um Objekte in einer Funktion zu verändern (Effekt!)

# Mutierende Funktionen

- Zeigerargumente können verwendet werden um Objekte in einer Funktion zu verändern (Effekt!)

## Beispiel

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

# Mutierende Funktionen

- Zeigerargumente können verwendet werden um Objekte in einer Funktion zu verändern (Effekt!)

## Beispiel

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



Übergabe der Adresse von a

Übergabe der Adresse des ersten Elementes hinter a

# Mutierende Funktionen

- Zeigerargumente können verwendet werden um Objekte in einer Funktion zu verändern (Effekt!)

## Beispiel

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



Übergabe der Adresse von a

Übergabe der Adresse des ersten Elementes hinter a

- Solche Funktionen heißen *mutierend*

# Mutierende Funktionen

- Zeigerargumente können verwendet werden um Objekte in einer Funktion zu verändern (Effekt!)

## Beispiel

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



Übergabe der Adresse von a

Übergabe der Adresse des ersten Elementes hinter a

- Solche Funktionen heissen *mutierend*

# Const-Korrektheit

- Es gibt auch *nicht* mutierende Funktionen, die auf Elemente eines Arrays zugreifen

# Const-Korrektheit

- Es gibt auch *nicht* mutierende Funktionen, die auf Elemente eines Arrays zugreifen

## Beispiel

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest of the values described by [begin , end)
// is returned
int min ( int* begin , int* end)
{
    assert (begin != end);
    int m = *begin; // current candidate for the minimum
    for (const int* ptr = begin+1; ptr != end; ++ptr)
        if (*ptr < m) m = *ptr;
    return m;
}
```

# Const-Korrektheit

- Es gibt auch *nicht* mutierende Funktionen, die auf Elemente eines Arrays zugreifen

## Beispiel

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest of the values described by [begin , end)
// is returned
int min (const int* begin ,const int* end)
{
    assert (begin != end);
    int m = *begin; // current candidate for the minimum
    for (const int* ptr = begin+1; ptr != end; ++ptr)
        if (*ptr < m) m = *ptr;
    return m;
}
```

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

# Const-Korrektheit

- Es gibt auch *nicht* mutierende Funktionen, die auf Elemente eines Arrays zugreifen

## Beispiel

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest of the values described by [begin , end)
// is returned
int min (const int* begin ,const int* end)
{
    assert (begin != end);
    int m = *begin; // current candidate for the minimum
    for (const int* ptr = begin+1; ptr != end; ++ptr)
        if (*ptr < m) m = *ptr;
    return m;
}
```

**const** bezieht sich auf **int**, nicht auf den Zeiger. Erläuterung nächste Folie

- 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.

# 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 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.

# Algorithmen

Für viele alltägliche Probleme existieren vorgefertigte Lösungen in der Standardbibliothek.

## Beispiel: Füllen eines Feldes

```
// Program: fill3.cpp
// use a standard algorithm to fill a range
#include <iostream >
#include <algorithm > // needed for std::fill

int main()
{
    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
    std::cout << "\n";

    return 0;
}
```

# Algorithmen

Für viele alltägliche Probleme existieren vorgefertigte Lösungen in der Standardbibliothek.

## Beispiel: Füllen eines Feldes

```
// Program: fill3.cpp
// use a standard algorithm to fill a range
#include <iostream >
#include <algorithm > // needed for std::fill

int main()
{
    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
    std::cout << "\n";

    return 0;
}
```

### Vorteile der Verwendung der Standardbibliothek

- Einfachere Implementation
- Fehlerquellen minimiert
- Guter, schneller Code
- Universeller Code (nächste Folie)

# Algorithmen

Die gleichen vorgefertigten Algorithmen existieren für viele Datentypen

## Beispiel: Füllen eines Feldes

```
// Program: fill4.cpp
// use a standard algorithm to fill a range
#include <iostream>
#include <string>
#include <algorithm> // needed for std::fill

int main()
{
    std::string a[5];
    std::fill (a, a+5, "bla");
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " "; // bla bla bla bla
    std::cout << "\n";

    return 0;
}
```

# Algorithmen

Die gleichen vorgefertigten Algorithmen existieren für viele Datentypen

## Beispiel: Füllen eines Feldes

```
// Program: fill4.cpp
// use a standard algorithm to fill a range
#include <iostream>
#include <string>
#include <algorithm> // needed for std::fill

int main()
{
    std::string a[5];
    std::fill (a, a+5, "bla");
    for (int i=0; i<5; ++i)
        std::cout << a[i] << " "; // bla bla bla bla
    std::cout << "\n";

    return 0;
}
```

Es existieren natürlich auch Algorithmen für schwierigere Probleme, wie z.B. das (effiziente) Sortieren eines Feldes

# Überladen von Funktionen

- Funktionen sind durch Ihren Namen im Gültigkeitsbereich ansprechbar
- Es ist sogar möglich, mehrere Funktionen des gleichen Namens zu definieren und zu deklarieren
- Die „richtige“ Version wird aufgrund der *Signatur* der Funktion ausgewählt

# Überladen von Funktionen

- Die Signatur einer Funktion ist bestimmt durch Art, Anzahl und Reihenfolge der Argumente

```
double sq(double x) { ... }           // fkt 1
int sq(int x) { ... }                 // fkt 2
int pow(int b, int e) { ... }         // fkt 3
int pow(int e) { return pow(2,e); }   // fkt 4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“ (wir vertiefen das nicht)

```
std::cout << sq(3);      // Compiler wählt Funktion 2
std::cout << sq(1.414); // Compiler wählt Funktion 1
std::cout << pow(2);    // Compiler wählt Funktion 4
std::cout << pow(3,3);  // Compiler wählt Funktion 3
```

# Überladen von Funktionen

Mit dem Überladen von Funktionen lassen sich also

- verschiedene Aufrufvarianten des gleichen Algorithmus realisieren und / oder
- verschiedene Algorithmen für verschiedene Datentypen mit dem gleichen Namen verbinden.

Funktioniert ein Algorithmus für verschiedene Datentypen identisch, so verwendet man in C++ *Templates*.

# Templates

- Templates erlauben die Spezifikation eines Typs als Argument.

## Beispiel: *fill* mit Templates

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
template <typename T>
void fill (T* begin , T* end, const T value) {
    for (T* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}

int main() {
    int a[5];
    fill(a, a+5, 1);      // ...
    std::string b[5];
    fill(b, b+5, "bla"); // ...
}
```

- Wir gehen vorerst nicht weiter darauf ein.

# Templates

- Templates erlauben die Spezifikation eines Typs als Argument.

## Beispiel: *fill* mit Templates

```
// PRE: [begin , end) is a valid range
// POST: *p is set to value , for p in [begin , end)
template <typename T>
void fill(T* begin ↑ T* end, const T value) {
    for (T* ptr = begin; ptr != end; ++ptr)
        *ptr = value;
}

int main() {
    int a[5];
    fill(a, a+5, 1);      // ...
    std::string b[5];
    fill(b, b+5, "bla"); // ...
}
```

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

- Wir gehen vorerst nicht weiter darauf ein.

# Vektor-Iteratoren

Annahme: wir wollen einen Vektor mit einem Wert füllen

- Der vorher diskutierte *fill*-Algorithmus mit Templates funktioniert dafür nicht.

# Vektor-Iteratoren

Annahme: wir wollen einen Vektor mit einem Wert füllen

- Der vorher diskutierte *fill*-Algorithmus mit Templates funktioniert dafür nicht.
- Der Grund: Vektoren verhalten sich zwar wie Felder, sind aber keine; man kann nicht mit Hilfe von Zeigern auf ihnen operieren.
- Trotzdem funktioniert `std::fill` auch für Vektoren, wie folgt:

# Vektor-Iteratoren

## Beispiel: *fill* auf einem Vektor

```
// Program: fill6.cpp
// use a standard algorithm to fill a vector
#include <iostream >
#include <vector >
#include <algorithm > // needed for std::fill

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

    return 0;
}
```

# Vektor-Iteratoren

## Beispiel: *fill* auf einem Vektor

```
// Program: fill16.cpp
// use a standard algorithm to fill a vector
#include <iostream >
#include <vector >
#include <algorithm > // needed for std::fill

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

    return 0;
}
```

Erklärung folgt

# Vektor-Iteratoren

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

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

für nicht-mutierenden Zugriff

`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

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

für mutierenden Zugriff

- Vektor-Iteratoren sind keine Zeiger, verhalten sich aber ebenso:

# Vektor-Iteratoren

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

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

für nicht-mutierenden Zugriff

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

für mutierenden Zugriff

- Vektor-Iteratoren sind keine Zeiger, verhalten sich aber ebenso:

# Vektor-Iteratoren

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

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

für nicht-mutierenden Zugriff

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

für mutierenden Zugriff

- Vektor-Iteratoren sind keine Zeiger, verhalten sich aber ebenso:
  - zeigen auf ein Vektor-Element und können dereferenziert werden

# Vektor-Iteratoren

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

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

für nicht-mutierenden Zugriff

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

für mutierenden Zugriff

- Vektor-Iteratoren sind keine Zeiger, verhalten sich aber ebenso:
  - zeigen auf ein Vektor-Element und können dereferenziert werden
  - können mit arithmetischen Ausdrücken verwendet werden wie Zeiger

# Vektor-Iteratoren

- Die Konversion Vektor  $\Rightarrow$  Iterator muss explizit gemacht werden

# Vektor-Iteratoren

- Die Konversion Vektor  $\Rightarrow$  Iterator muss explizit gemacht werden
  - `v.begin()` zeigt auf das erste Element von `v`
  - `v.end()` zeigt auf das Element nach dem letzten Element von `v`

# Vektor-Iteratoren

- Die Konversion Vektor  $\Rightarrow$  Iterator muss explizit gemacht werden
  - **v.begin()** zeigt auf das erste Element von **v**
  - **v.end()** zeigt auf das Element nach dem letzten Element von **v**
- Damit können wir einen Vektor traversieren

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

# Vektor-Iteratoren

- Die Konversion Vektor  $\Rightarrow$  Iterator muss explizit gemacht werden
  - **v.begin()** zeigt auf das erste Element von **v**
  - **v.end()** zeigt auf das Element nach dem letzten Element von **v**
- Damit können wir einen Vektor traversieren

```
// output all elements of v, using iteration
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 Programmbeispiel

```
// Program: vector_iterators.cpp
// demonstrates vector iterators
#include<iostream>
#include<vector>

typedef std::vector<int>::const_iterator Cvit;
typedef std::vector<int>::iterator Vit;

int main()
{
    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 << " ";
    std::cout << "\n";

    // 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.begin()[i] << " "; // complicated way of writing v[i]
    std::cout << "\n";

    return 0;
}
```

# Vektor-Iteratoren Programmbeispiel

```
// Program: vector_iterators.cpp
// demonstrates vector iterators
#include<iostream>
#include<vector>

typedef std::vector<int>::const_iterator Cvit;
typedef std::vector<int>::iterator Vit;

int main()
{
    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 << " ";
    std::cout << "\n";

    // 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.begin()[i] << " "; // complicated way of writing v[i]
    std::cout << "\n";

    return 0;
}
```

Typdefinition führt neuen Namen als Abkürzung für Typ ein. Erklärung folgt.

Vektor-Element auf das `it` verweist

Inkrementieren des Iterators

Kurzschreibweise für  
`* (v.begin() + i)`

# Typdefinitionen

- Insbesondere bei komplizierten Typen kann die repetitive Verwendung des gleichen Typs umständlich sein

# Typdefinitionen

- Insbesondere bei komplizierten Typen kann die repetitive Verwendung des gleichen Typs umständlich sein
- Dann hilft die Möglichkeit der Deklaration eines *Typ-Alias* mit

`typedef Typ Name`

bestehender Typ

Name, unter dem der Typ neu auch angesprochen werden kann

# Typdefinitionen

- Insbesondere bei komplizierten Typen kann die repetitive Verwendung des gleichen Typs umständlich sein
- Dann hilft die Möglichkeit der Deklaration eines *Typ-Alias* mit

**typedef Typ Name**

## Beispiele

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

# Container und Iteratoren

- Das Traversieren von Daten ist ein generisches Verfahren, das allgemein auf *Container* angewendet werden kann.

# Container und Iteratoren

- Das Traversieren von Daten ist ein generisches Verfahren, das allgemein auf *Container* angewendet werden kann.
- Ein Container ist ein Objekt, welches
  - Elemente eines zugrundeliegenden Typs speichern kann

# Container und Iteratoren

- Das Traversieren von Daten ist ein generisches Verfahren, das allgemein auf *Container* angewendet werden kann.
- Ein Container ist ein Objekt, welches
  - Elemente eines zugrundeliegenden Typs speichern kann
  - mindestens die Möglichkeit zur Traversierung aller seiner Elemente anbietet

# Container und Iteratoren

- Das Traversieren von Daten ist ein generisches Verfahren, das allgemein auf *Container* angewendet werden kann.
- Ein Container ist ein Objekt, welches
  - Elemente eines zugrundeliegenden Typs speichern kann
  - mindestens die Möglichkeit zur Traversierung aller seiner Elemente anbietet
- Felder und Vektoren sind solche Container

# Container und Iteratoren

- Das Traversieren von Daten ist ein generisches Verfahren, das allgemein auf *Container* angewendet werden kann.
- Ein Container ist ein Objekt, welches
  - Elemente eines zugrundeliegenden Typs speichern kann
  - mindestens die Möglichkeit zur Traversierung aller seiner Elemente anbietet
- Felder und Vektoren sind solche Container

Bei Feldern und Vektoren wird Traversierung über Iteratoren / Zeiger (natürlich) oder durch wahlfreien Zugriff angeboten

# Container und Iteratoren

- Das Traversieren von Daten ist ein generisches Verfahren, das allgemein auf *Container* angewendet werden kann.
- Ein Container ist ein Objekt, welches
  - Elemente eines zugrundeliegenden Typs speichern kann
  - mindestens die Möglichkeit zur Traversierung aller seiner Elemente anbietet
- Felder und Vektoren sind solche Container
- Andere Beispiele: Mengen (Sets)

# Mengen (Sets)

- Eine Menge ist eine ungeordnete Zusammenfassung von Elementen, wobei jedes Element nur einmal vorkommen kann

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

# Mengen (Sets)

- Eine Menge ist eine ungeordnete Zusammenfassung von Elementen, wobei jedes Element nur einmal vorkommen kann

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

- In C++ repräsentiert der Typ `std::set<T>` eine Menge von 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!

# Mengen: Beispiel einer Anwendung

- Fasse den Text als Menge von Buchstaben auf:

```
// Prog: set.cpp
// demonstrates usage of the std::set container
#include<iostream>
#include<set>
#include<algorithm>

typedef std::set<char>::const_iterator Sit;

int main()
{
    std::string text =
        "What are the distinct characters in this string?";
    std::set<char> s (text.begin(), text.end());
    ...
}
```

# Mengen: Beispiel einer Anwendung

- Fasse den Text als Menge von Buchstaben auf:

```
// Prog: set.cpp
// demonstrates usage of the std::set container
#include<iostream>
#include<set>
#include<algorithm>

typedef std::set<char>::const_iterator Sit;

int main()
{
    std::string text =
        "What are the distinct characters in this string?";
    std::set<char> s (text.begin(), text.end());
    ...
}
```

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

# Mengen: Beispiel einer Anwendung

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

...

```
// 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 (SIt it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
std::cout << "\n";

return 0;
}
```

# Mengen: Beispiel einer Anwendung

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

...

```
// 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 (SIt it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
std::cout << "\n";

return 0;
}
```

Generischer Suchalgorithmus,  
aufrufbar mit Iterator-Bereich

# Mengen: Beispiel einer Anwendung

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

```
...
// 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 (SIt it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
std::cout << "\n";
return 0;
}
```

Generischer Suchalgorithmus,  
aufrufbar mit Iterator-Bereich

Ausgabe:  
Good question!  
?Wacdeghinrst

# Wahlfreier Zugriff auf Mengen?

- Kann man auf Mengen mit wahlfreiem Zugriff operieren?

# Wahlfreier Zugriff auf Mengen?

- Kann man auf Mengen mit wahlfreiem Zugriff operieren?

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i]; // or s.begin() [i]
std::cout << "\n";
```

# Wahlfreier Zugriff auf Mengen?

- Kann man auf Mengen mit wahlfreiem Zugriff operieren?

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i]; // or s.begin()[i]
std::cout << "\n";
```



Fehlermeldung: no subscript operator

- Nein, Mengen sind ungeordnet und unterstützen keinen wahlfreien Zugriff. Es gibt kein “*i*-tes Element”.

# Das Konzept der Iteratoren

- C++ kennt verschiedene Iterator-Typen

# Das Konzept der Iteratoren

- C++ kennt verschiedene Iterator-Typen
  - Alle unterstützen den Dereferenz-Operator `*` und traversieren mit `++`
  - Manche unterstützen mehr, z.B. wahlfreien Zugriff oder rückwärts iterieren mit `--`

# Das Konzept der Iteratoren

- C++ kennt verschiedene Iterator-Typen
  - Alle unterstützen den Dereferenz-Operator `*` und traversieren mit `++`
  - Manche unterstützen mehr, z.B. wahlfreien Zugriff oder rückwärts iterieren mit `--`
- Jeder Container-verarbeitende Algorithmus der Standardbibliothek ist *generisch*, funktioniert also für alle Container, deren Iteratoren bestimmte Anforderungen erfüllen

# Das Konzept der Iteratoren

- C++ kennt verschiedene Iterator-Typen
  - Alle unterstützen den Dereferenz-Operator `*` und traversieren mit `++`
  - Manche unterstützen mehr, z.B. wahlfreien Zugriff oder rückwärts iterieren mit `--`
- Jeder Container-verarbeitende Algorithmus der Standardbibliothek ist *generisch*, funktioniert also für alle Container, deren Iteratoren bestimmte Anforderungen erfüllen
- `std::find` erfordert z.B. nur `*` und `++`

# Das Konzept der Iteratoren

- C++ kennt verschiedene Iterator-Typen
  - Alle unterstützen den Dereferenz-Operator `*` und traversieren mit `++`
  - Manche unterstützen mehr, z.B. wahlfreien Zugriff oder rückwärts iterieren mit `--`
- Jeder Container-verarbeitende Algorithmus der Standardbibliothek ist *generisch*, funktioniert also für alle Container, deren Iteratoren bestimmte Anforderungen erfüllen
  - Weitere Details des Container-Typs sind für den Algorithmus 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;
```

# 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  $[\text{begin}, \text{end})$  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!