Zeiger und Iteratoren

Zeiger (auf Array)	**Iterieren** über ein Array

Diese Befehle gelten **zusätzlich** zu denen unter Zeiger (generell) (siehe Summary 8), **falls** Zeiger auf einem **Array** verwendet werden.

Wichtige Befehle (gelte `int a[6];`):

Zeiger auf `a[0]`:	`int* ptr = a; // Works ONLY if a is ARRAY!`
temporärer Shift:	`ptr + 3`
	`ptr - 3`
permanenter Shift:	`++ptr`
	`ptr++`
	`--ptr`
	`ptr--`
	`ptr += 3`
	`ptr -= 3`
Distanz bestimmen:	`ptr1 - ptr2`
Position vergleichen:	`ptr1 < ptr2` (Sonst: `<=, >, >=, ==, !=`)

Die sogenannte Array-to-Pointer-Conversion erlaubt es, einen (temporären) Zeiger auf das Element beim Index `0` ganz einfach zu bekommen. Beispiele: `int* ptr = a;` oder `a + 3`

Achtung: Die grünen Shifts erzeugen einen neuen (temporären) Zeiger und verschieben `ptr` **nicht**. Die violetten Shifts verschieben aber `ptr`.

Achtung: Der Programmierer ist *selbst* dafür verantwortlich, dass Zeiger das Array nicht verlassen. (z.B. `ptr - 1` soll vermieden werden, falls `ptr` auf `a[0]` zeigt). Die einzige erlaubte Ausnahme ist der Past-the-End-Zeiger, der aber **nicht dereferenziert** werden darf.

```
// Read 6 values into an array
std::cout << "Enter 6 numbers:\n";
int a[6];
int* pTE = a+6;
for (int* i = a; i < pTE; ++i)
    std::cin >> *i; // read into array element

// Output:   a[0]+a[3],  a[1]+a[4],  a[2]+a[5]
for (int* i = a; i < a+3; ++i) {
    assert(i+3 < pTE); // Assert that i+3 stays inside.
    std::cout << (*i + *(i+3)) << ",  ";
}
```

`const` (Zeiger)	**kein Schreibzugriff** auf das Objekt

```
int a = 5;
int b = 8;

const int* ptr = &a;
*ptr = 3; // NOT valid (write to object)
ptr = &b; // valid (write to pointer (i.e. switch object))
```

Iterator (auf Vektor)	**Iterieren** über einen Vektor.

Im Folgenden wird nur auf die Unterschiede zum Zeiger (auf Array) eingegangen. Die restliche Bedienung erfolgt gleich.

Erfordert: `#include<vector>`

Wichtige Befehle (gelte `std::vector<int> a (6, 0);`):

Definition: `std::vector<int>::iterator itr = ...;`
Iterator auf `a[0]`: `a.begin()`
Past-the-End-Iterator: `a.end()`

Anstelle des `...` in der Definition eines Iterators müssen andere Iteratoren stehen (z.B. `a.begin()`).

Vektoren unterstützen **keine *automatische* Konvertierung** zu einem Iterator auf das Element mit Index `0`:

```
int* ptr = arr; // Works ONLY if arr is ARRAY!
std::vector<int>::iterator itr = vec.begin(); // for VECTORS
```

Dafür haben Vektoren im Gegensatz zu Pointern einen bequemen Schnellzugriff auf den Past-the-End-Iterator: `a.end()`

(...)

(...)

```
// Same example as for arrays, but now for vectors.
// To avoid the lengthy lines see entry on typedef.

// Read 6 values into a vector
std::cout << "Enter 6 numbers:\n";
std::vector<int> a (6, 0);
for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into object of iterator

// Output:   a[0]+a[3],  a[1]+a[4],  a[2]+a[5]
for (std::vector<int>::iterator i = a.begin(); i < a.begin()+3; ++i) {
    assert(i+3 < a.end()); // Assert that i+3 stays inside.
    std::cout << (*i + *(i+3)) << ",  ";
}
```

`const` (Iterator)	**kein Schreibzugriff** auf das Objekt

Vorsicht: Einen `const`-Iterator erzeugt man mittels
`std::vector<int>::const_iterator ...`
und **nicht** mittels
`const std::vector<int>::iterator ...`

Die zweite Version erzeugt einen Iterator, den man nicht herumschieben kann. In dieser Vorlesung gehen wir aber nur auf die Iteratoren näher ein, welche den Schreibzugriff auf *das Objekt* verbieten (erste Variante oben).

```
std::vector<int> a (6, -8); // a is: -8 -8 -8 -8 -8 -8

std::vector<int>::const_iterator itr = a.begin() + 3;
*itr = 4;        // NOT valid
itr = a.begin(); // valid (itr now points to a[0])
```

Datentypen

`typedef old new;`	Lange **Datentyp-Namen verkürzen**.

```
// Same example as for vectors, but now using typedef:
typedef std::vector<int> Vec;
typedef std::vector<int>::iterator Vit;

// Read 6 values into a vector
std::cout << "Enter 6 numbers:\n";
Vec a (6, 0);
for (Vit i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into object of iterator

// Output:   a[0]+a[3],  a[1]+a[4],  a[2]+a[5]
for (Vit i = a.begin(); i < a.begin()+3; ++i) {
    assert(i+3 < a.end()); // Assert that i+3 stays inside.
    std::cout << (*i + *(i+3)) << ",  ";
}
```

Standard-Funktionen auf Arrays, Vektoren, ...

`std::fill(b, p, val)`	Wert `val` in einen **Bereich** `[b,p)` **einlesen**
Erfordert: `#include<algorithm>`	

```
// Goal: Generate vector: 4 4 4 2 2
std::vector<int> vec (5, 4);               // vec: 4 4 4 4 4
std::fill(vec.begin()+3, vec.end(), 2);    // vec: 4 4 4 2 2
```

`std::find(b, p, val)`	`val` **suchen** im Bereich `[b,p)`

(...)

(...)

Erfordert: `#include<algorithm>`

Zurückgegeben wird ein **Iterator** auf das *erste* gefundene Vorkommnis.

Wenn `std::find` nicht fündig wird, gibt es den Past-the-End-Iterator `p` zurück. (Beachte: Past-the-End ist bezüglich Bereich `[b,p)` gemeint.)

```
typedef std::vector<int>::iterator Vit;
std::vector<int> vec (5, 2);
vec[3] = -7

// Goal: Find index of -7 in vec: 2 2 2 -7 2
Vit pos_itr = std::find(vec.begin(), vec.end(), -7);
std::cout << (pos_itr - vec.begin()) << "\n"; // Output: 3
```

`std::min_element(b, p)`	**Iterator auf Minimum** im Bereich `[b,p)`

Erfordert: `#include<algorithm>`

Wenn das Minimum nicht eindeutig ist, so wird ein Iterator auf das erste Vorkommnis zurückgegeben.

```
// Goal: Make sure that all inputs are > 0
std::vector<int> vec (10, 0);
for (int i = 0; i < 10; ++i)
    std::cin >> vec[i];

assert( *std::min_element(vec.begin(), vec.end()) > 0 );
    // Note: We have to dereference the (r-value-)iterator.
```

Operatoren

`&`	**Adressoperator** (siehe: *Adresse auslesen* unter `Zeiger (generell)`, Summary 8)

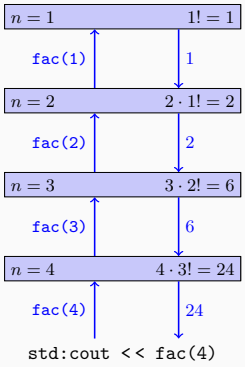
Präzedenz: 16 und Assoziativität: `rechts`

`*`	**Dereferenz-Operator** (siehe: *Zugriff auf Objekt* unter `Zeiger (generell)`)
Präzedenz: 16 und Assoziativität: `rechts`	

Funktionen

`Rekursion`	**Selbstaufruf** einer Funktion

Jeder rekursive Funktionsaufruf hat seine eigenen, unabhängigen Variablen und Argumente. Dies kann man sich sehr gut anhand des in der Vorlesung gezeigten Stacks vorstellen (`fac` ist im Beispiel unten definiert):




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