

Zeiger und Iteratoren

Zeiger (generell)	Adresse eines Objekts im Speicher								
<p>Wichtige Befehle:</p> <p>Definition: <code>int* ptr = address_of_type_int;</code> (ohne Startwert: <code>int* ptr = 0;</code>)</p> <p>Zugriff auf Zeiger: <code>ptr = otr_ptr // Pointer gets new target.</code></p> <p>Zugriff auf Target: <code>*ptr = 5 // Target gets new value 5.</code></p> <p>Adresse auslesen: <code>int* ptr_to_a = &a; // (a is int-variable)</code></p> <p>Vergleich: <code>ptr == otr_ptr // Same target?</code> <code>ptr != otr_ptr // Different targets?</code></p> <p>(Anstatt <code>int</code> gehen natürlich auch andere Typen.) (Eine <code>address_of_type_int</code> kann man durch einen anderen Zeiger oder auch mittels dem Adressoperator <code>&</code> erzeugen (siehe Beispiel unten).)</p> <p>Der Wert des Zeigers ist die Speicheradresse des Targets. Will man also das Target via diesen Zeiger verändern, muss man zuerst "zu der Adresse gehen". Genau das macht der Dereferenz-Operator <code>*</code>.</p> <p>Beispiel: (Gelte <code>int a = 5;</code>)</p> <table><tr><td>Wert von <code>a</code>:</td><td>5</td></tr><tr><td>Speicheradresse von <code>a</code>:</td><td>0x28fef8</td></tr><tr><td>Wert von <code>a_ptr</code>:</td><td>0x28fef8</td></tr><tr><td>Wert von <code>*a_ptr</code>:</td><td>5</td></tr></table> <p>Ein Zeiger kann immer nur auf den entsprechenden Typ zeigen. (z.B. <code>int* ptr = &a;</code> Hier muss <code>a</code> Typ <code>int</code> haben.)</p>		Wert von <code>a</code> :	5	Speicheradresse von <code>a</code> :	0x28fef8	Wert von <code>a_ptr</code> :	0x28fef8	Wert von <code>*a_ptr</code> :	5
Wert von <code>a</code> :	5								
Speicheradresse von <code>a</code> :	0x28fef8								
Wert von <code>a_ptr</code> :	0x28fef8								
Wert von <code>*a_ptr</code> :	5								
<pre>int a = 5; int* a_ptr = &a; // a_ptr points to a a_ptr = a; // NOT valid (same as: a_ptr = 5;) // 5 is NOT an address. a_ptr = &a; // valid *a_ptr = 9; // a obtains value 9 std::cout << "a == " << a << "\n"; // Output: a == 9 std::cout << "a == " << *a_ptr << "\n"; // Output: a == 9</pre>									

Programmier-Befehle - Woche 09

<code>const</code> (Zeiger)	kein Schreibzugriff auf das Target
<pre>int a = 5; int b = 8; const int* ptr = &a; *ptr = 3; // NOT valid (write to target) ptr = &b; // valid (write to pointer (i.e. switch target))</pre>	

Zeiger (auf Array)	Iterieren über ein Array										
<p>Diese Befehle gelten zusätzlich zu denen unter <code>Zeiger (generell)</code>, falls Zeiger auf einem Array verwendet werden.</p> <p>Wichtige Befehle (gelte <code>int a[6];</code>):</p> <table><tr><td>Zeiger auf a[0]:</td><td><code>int* ptr = a; // Works ONLY if a is ARRAY!</code></td></tr><tr><td>temporärer Shift:</td><td><code>ptr + 3</code> <code>ptr - 3</code></td></tr><tr><td>permanentener Shift:</td><td><code>++ptr</code> <code>ptr++</code> <code>--ptr</code> <code>ptr--</code> <code>ptr_1 += 3</code> <code>ptr_1 -= 3</code></td></tr><tr><td>Distanz bestimmen:</td><td><code>ptr1 - ptr2</code></td></tr><tr><td>Position vergleichen:</td><td><code>ptr1 < ptr2</code> (Sonst: <code><=</code>, <code>></code>, <code>>=</code>, <code>==</code>, <code>!=</code>)</td></tr></table> <p>Die sogenannte Array-to-Pointer-Conversion erlaubt es, einen (temporären) Zeiger auf das Element beim Index 0 ganz einfach zu bekommen. Beispiele: <code>int* ptr = a;</code> oder <code>a + 3</code></p> <p>Achtung: Die grünen Shifts erzeugen einen neuen (temporären) Zeiger und verschieben <code>ptr</code> nicht. Die violetten Shifts verschieben aber <code>ptr</code> und geben eine Referenz auf ihn zurück. So ist beispielsweise Folgendes möglich: <code>++(++ptr)</code></p> <p>Achtung: Der Programmierer ist <i>selbst</i> dafür verantwortlich, dass Zeiger das Array nicht verlassen. (z.B. <code>ptr - 1</code> soll vermieden werden, falls <code>ptr</code> auf <code>a[0]</code> zeigt). Die einzige erlaubte Ausnahme ist der Past-the-End-Zeiger, der aber nicht dereferenziert werden darf.</p>		Zeiger auf a[0]:	<code>int* ptr = a; // Works ONLY if a is ARRAY!</code>	temporärer Shift:	<code>ptr + 3</code> <code>ptr - 3</code>	permanentener Shift:	<code>++ptr</code> <code>ptr++</code> <code>--ptr</code> <code>ptr--</code> <code>ptr_1 += 3</code> <code>ptr_1 -= 3</code>	Distanz bestimmen:	<code>ptr1 - ptr2</code>	Position vergleichen:	<code>ptr1 < ptr2</code> (Sonst: <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>!=</code>)
Zeiger auf a[0]:	<code>int* ptr = a; // Works ONLY if a is ARRAY!</code>										
temporärer Shift:	<code>ptr + 3</code> <code>ptr - 3</code>										
permanentener Shift:	<code>++ptr</code> <code>ptr++</code> <code>--ptr</code> <code>ptr--</code> <code>ptr_1 += 3</code> <code>ptr_1 -= 3</code>										
Distanz bestimmen:	<code>ptr1 - ptr2</code>										
Position vergleichen:	<code>ptr1 < ptr2</code> (Sonst: <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>!=</code>)										

(...)

(...)

```
// 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 target

// 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 < 0); // Assert that i+3 stays inside.
                                // Same effect:  assert(i+3 < pTE);
    int sum = *i + *(i+3);
    std::cout << sum << ", ";
}
}
```

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);`):

Definition: `std::vector<int>::iterator itr`
`= itr_of_type_int;`

Iterator auf a[0]: `a.begin()`

Past-the-End-Iterator: `a.end()`

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> itr = vec.begin(); // Counterpart for VECTORS`

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

(...)

Programmier-Befehle - Woche 09

(...)

```
// 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);
for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into target 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() < 0); // Assert that i+3 stays inside.
                                // Same effect:  assert(i+3 < a.end());
    int sum = *i + *(i+3);
    std::cout << sum << ", ";
}
}
```

`const` (Iterator)

kein Schreibzugriff auf das Target

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 Target 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

<code>typedef old new;</code>	Lange Datentyp-Namen verkürzen.
<pre>// Same example as for vectors, but now using typedef: typedef std::vector<int> Vec; typedef std::vector<int>::iterator It; // Read 6 values into a vector std::cout << "Enter 6 numbers:\n"; Vec a (6); for (It i = a.begin(); i < a.end(); ++i) std::cin >> *i; // read into target of iterator // Output: a[0]+a[3], a[1]+a[4], a[2]+a[5] for (It i = a.begin(); i < a.begin()+3; ++i) { assert((i+3) - a.end() < 0); // Assert that i+3 stays inside. // Same effect: assert(i+3 < a.end()); int sum = *i + *(i+3); std::cout << sum << ", "; } }</pre>	

Vektoren

<code>...push_back(v)</code>	Zusätzliches Element v hinten anhängen
<p>Erfordert: <code>#include <vector></code></p> <p><code>push_back</code> ist sehr stark, wenn man eine unbekannte Anzahl Inputs einlesen soll.</p>	
<pre>// General functionality std::vector<int> my_vec (5, 0); // my_vec: 0 0 0 0 0 my_vec.push_back(7); // my_vec: 0 0 0 0 0 7 // Read unknown number of inputs std::vector<int> input_container (0); // vector of length 0 int i; while (std::cin >> i) input_container.push_back(i);</pre>	

Standard-Funktionen auf Arrays, Vektoren, Sets, Strings, ...

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

<code>std::find(b, p, val)</code>	<code>val</code> suchen im Bereich <code>[b,p)</code>
Erfordert: <code>#include <algorithm></code>	
Zurückgegeben wird ein Iterator auf das <i>erste</i> gefundene Vorkommnis.	
Wenn <code>std::find</code> nicht fündig wird, gibt es den Past-the-End-Iterator <code>p</code> zurück. (Beachte: Past-the-End ist bezüglich Bereich <code>[b,p)</code> gemeint.)	
<pre>typedef std::vector<int>::iterator It; std::vector<int> vec = {8, 1, 0, -7, 7}; // Goal: Find index of -7 in vec: 8 1 0 -7 7 It pos_itr = std::find(vec.begin(), vec.end(), -7); std::cout << (pos_itr - vec.begin()) << "\n"; // Output: 3</pre>	

Programmier-Befehle - Woche 09

<code>std::sort(b, e)</code>	Bereich [b, e) sortieren
Erfordert: <code>#include <algorithm></code>	
<code>std::sort</code> funktioniert nur, wenn Random-Access Iteratoren für b und e übergeben werden. Somit funktioniert <code>std::sort</code> z.B. für Felder und Vektoren, aber nicht z.B. für Sets.	
<pre>std::vector<int> vec = {8, 1, 0, -7, 7}; std::sort(vec.begin(), vec.end()); // vec: -7 0 1 7 8</pre>	

<code>std::min_element(b, p)</code>	Iterator auf Minimum im Bereich [b,p)
Erfordert: <code>#include <algorithm></code>	
Wenn das Minimum nicht eindeutig ist, so wird ein Iterator auf das erste Vorkommen zurückgegeben.	
<pre>// Goal: Make sure that all inputs are > 0 std::vector<int> vec; int i; while (std::cin >> i) vec.push_back(i); assert(*std::min_element(vec.begin(), vec.end()) > 0); // Note: We have to dereference the (r-value-)iterator.</pre>	

Operatoren

&	Adressoperator siehe: Adresse auslesen (unter Zeiger (generell))
Präzedenz: 16 und Assoziativität: rechts	

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