

Datentypen

Array (mehrdim.)	mehrdimensionale "Massenvariable" eines bestimmten Typs
Wichtige Befehle:	
Definition:	<code>int my_arr[2][3] = { {2, 1, 6}, {8, -1, 4} };</code>
Zugriff:	<code>my_arr[1][1] = 8 * my_arr[0][2];</code>
(Anstatt <code>int</code> gehen natürlich auch andere Typen.) (Die Definition kann auch ohne Initialisierung erfolgen: <code>int my_arr[2][3];</code>)	
<pre>int my_arr[2][3] = { {2, 1, 6}, {8, -1, 4} }; my_arr[1][1] = 8 * my_arr[0][2]; // my_arr becomes // 2, 1, 6 // 8, 48, 4</pre>	

Vektoren (mehrdim.)	komfortabler mehrdimensionaler Array eines bestimmten Typs
Erfordert: <code>#include <vector></code>	
Wichtige Befehle:	
Definition:	<code>std::vector<std::vector<int> ></code> <code>my_vec (n_rows, std::vector<int>(n_cols,</code> <code>init_value))</code>
Zugriff:	(wie mehrdim. Array)
(Anstatt <code>int</code> gehen natürlich auch andere Typen.) (Die Definition kann auch ohne Initialisierung erfolgen: <code>std::vector<std::vector<int> ></code> <code>my_vec (n_rows, std::vector<int>(n_cols))</code>)	
Bei ganz neuen Versionen von C++ müssen die <code>> ></code> nicht mehr zwingend einen Abstand dazwischen haben. Allerdings ist der Code beim Weglassen wegen einem eher kleinen Detail nicht mehr kompatibel mit älteren Standards.	

(...)

Programmier-Befehle - Woche 8

(...)

```
std::vector<std::vector<int> > my_vec (2, std::vector<int>(4, 0));
my_vec[1][2] = 3;
// my_vec becomes
// 0, 0, 0, 0
// 0, 0, 3, 0
```

`std::string`

komfortablerer Datentyp für Zeichen

Erfordert: `#include <string>`

Vorteile gegenüber char-Arrays:

variable Länge:	<code>std::string my_str (n, 'a');</code> (n kann variabel sein)
Länge abfragen:	<code>my_str.length()</code>
vergleichbar:	<code>text1 == text2</code>
hintereinander hängen:	<code>text1 += text2</code>
bequemer Output:	<code>std::cout << my_str;</code>



```
std::string my_word (5, 'a'); // initialize my_word as aaaaa
std::string ref (5, 'z');
my_word += ref; // append ref to my_word.
// Afterwards my_word: aaaaazzzzz
// Afterwards ref: zzzzz
std::cout << my_word.length() << "\n"; // output: 10
my_word[3] = 'b'; // change my_word to aaabzzzzz
if (my_word == ref) // false, already lengths differ (5 vs. 10)
    std::cout << "not output\n";
std::cout << my_word << "\n"; // output whole string at once
```

Input/Output

<code>std::noskipws</code>	Whitespaces einlesen
Erfordert: <code>#include <ios></code> oder <code>#include <iostream></code>	
<pre>char c; // Version 1: Assume the user enters: // a b std::cin >> c; // read 'a' std::cin >> c; // read 'b' // Version 2: Assume the user enters again: // a b std::cin >> std::noskipws; std::cin >> c; // read 'a' std::cin >> c; // read ' ' std::cin >> c; // read ' ' std::cin >> c; // read 'b'</pre>	

leerer Eingabestrom	Prüfe, ob mehr Eingaben vorhanden sind.
Dahinter steckt eine Konvertierung von <code>std::cin</code> zu <code>bool</code> :	
<pre>true: weitere Eingaben vorhanden false: keine Eingaben mehr vorhanden</pre>	
Wir brauchen diese Abfrage meistens, um eine Schleife solange laufen zu lassen, wie weitere Eingaben vorhanden sind. (siehe Beispiel unten)	
Erfolgt die Eingabe per Tastatur, so kann die Eingabe durch drücken von [Ctrl]+[D] beendet werden.	
<pre>char input; int length_of_text = 0; while(std::cin >> input) ++length_of_text; std::cout << length_of_text;</pre>	

Turtle

Turtle Plots	Zeichnen von Geraden														
<p>Erfordert: <code>#include "turtle.cpp"</code></p> <p>Die Turtle kennt 7 Befehle:</p> <table><tr><td><code>turtle::forward():</code></td><td>gezeichneter Schritt vorwärts</td></tr><tr><td><code>turtle::jump():</code></td><td>nicht gezeichneter Schritt vorwärts</td></tr><tr><td><code>turtle::left(my_angle):</code></td><td>Drehung nach links um 45 Grad</td></tr><tr><td><code>turtle::right(my_angle):</code></td><td>Drehung nach rechts um 45 Grad</td></tr><tr><td><code>turtle::save():</code></td><td>Position <i>und Blickrichtung</i> merken</td></tr><tr><td><code>turtle::restore():</code></td><td>Position <i>und Blickrichtung</i> laden</td></tr><tr><td><code>turtle::colorcycle():</code></td><td>Farbe wechseln (sehr feinstufig)</td></tr></table> <p>Die Turtle kann mehrere Positionen speichern (mittels <code>turtle::save()</code>). <code>turtle::restore()</code> lädt dann die Neueste und entfernt diese Position aus der Merkliste (somit ist dann die vorher zweitneuste Position neu die neuste).</p> <p>Die Dateien <code>turtle.cpp</code> und <code>bitmap.cpp</code> (siehe Vorlesungswebsite) müssen im gleichen Ordner liegen wie das eigene Programm.</p>		<code>turtle::forward():</code>	gezeichneter Schritt vorwärts	<code>turtle::jump():</code>	nicht gezeichneter Schritt vorwärts	<code>turtle::left(my_angle):</code>	Drehung nach links um 45 Grad	<code>turtle::right(my_angle):</code>	Drehung nach rechts um 45 Grad	<code>turtle::save():</code>	Position <i>und Blickrichtung</i> merken	<code>turtle::restore():</code>	Position <i>und Blickrichtung</i> laden	<code>turtle::colorcycle():</code>	Farbe wechseln (sehr feinstufig)
<code>turtle::forward():</code>	gezeichneter Schritt vorwärts														
<code>turtle::jump():</code>	nicht gezeichneter Schritt vorwärts														
<code>turtle::left(my_angle):</code>	Drehung nach links um 45 Grad														
<code>turtle::right(my_angle):</code>	Drehung nach rechts um 45 Grad														
<code>turtle::save():</code>	Position <i>und Blickrichtung</i> merken														
<code>turtle::restore():</code>	Position <i>und Blickrichtung</i> laden														
<code>turtle::colorcycle():</code>	Farbe wechseln (sehr feinstufig)														
<pre>// Draw a triangle (see below) turtle::forward(); turtle::left(120); turtle::forward(); turtle::left(120); turtle::forward(); // Move to neutral position turtle::left(120); // horizontal viewing direction turtle::jump(10); // move away from triangle (without drawing) // Draw a letter T (see below) turtle::forward(); turtle::save(); // memorize middle of letter T turtle::forward(); turtle::restore(); // go back to middle of letter T turtle::right(90); turtle::forward(2); // The argument means: 2 steps forward</pre> <hr/> <div style="display: flex; justify-content: space-around; align-items: center;"></div>															

Zeiger

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 object.</code></p> <p>Zugriff auf Objekt: <code>*ptr = 5 // Object 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 object?</code> <code>ptr != otr_ptr // Different objectss?</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 Objekts, auf das er zeigt. Will man also das Objekt 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 and NOT an array. a_ptr = &a; // valid *a_ptr = 9; // a obtains value 9 std::cout << a << "\n"; // Output: 9 std::cout << *a_ptr << "\n"; // Output: 9</pre>									

Programmier-Befehle - Woche 8

Zeiger (auf Array)	Iterieren über ein Array										
<p>Diese Befehle gelten zusätzlich zu denen unter Zeiger (generell), 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>permanenter 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>	permanenter 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>										
permanenter 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>)										
<pre>// 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 object // 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); const int sum = *i + *(i+3); std::cout << sum << ", "; } }</pre>											