

## Datentypen

<code>struct</code>	Container für Datentypen
Wichtige Befehle:	
<b>Definition:</b>	<pre>struct str_name {     int mem1;     bool mem2;     int mem3; };</pre>
<b>Objekt erstellen:</b>	<pre>str_name obj1;</pre>
<b>mit Startwerten:</b>	<pre>str_name obj2 = {3, true, 4};</pre>
<b>aus anderem Objekt:</b>	<pre>str_name obj3 = obj2;</pre>
<b>Zugriff auf Member:</b>	<pre>obj1.mem1</pre>
(Anstatt <code>int</code> und <code>bool</code> können die Member beliebige Typen haben.)	
Die <i>Definition</i> eines Structs hat ein <code>;</code> am Schluss.	
Nur der Zuweisungsoperator ( <code>=</code> ) wird automatisch erstellt (und kopiert dann die Member einzeln). Die anderen Operatoren (z.B. <code>==</code> , <code>!=</code> , ...) muss man selbst passend überladen (siehe Eintrag <a href="#">operator...</a> ). Den Zuweisungsoperator ( <code>=</code> ) kann man ebenfalls überladen, falls kein mitgliedswises Kopieren gewünscht ist.	
Als Struct-Member können Arrays kopiert werden. Sie werden standardmässig eintragsweise kopiert.	
Bei der <b>Default-Initialisierung</b> eines Objekts des Typs <code>str_name</code> werden alle Member einzeln default-initialisiert. Für fundamentale Typen ( <code>int</code> , <code>float</code> , usw.) bedeutet das, dass sie <i>uninitialisiert</i> sind, bis man ihnen nachträglich einen Wert zuweist. Das führt zu Problemen, falls man ihren <b>Wert vorher schon ausliest</b> .	

( ... )

# Programmier-Befehle - Woche 11

( ... )

```
struct candidate {
    std::string name;    // Name of the participant
    unsigned int height; // Her/his height
    int age;            // Her/his age
};

int main () {
    // initialization
    candidate mary;    // default-initialisation
    std::cout << mary.height; // Undefined behaviour
    mary.name = "Mary"; mary.height = 168; mary.age = 43;
    std::cout << mary.height; // Problem gone: mary.height is 168
    candidate bob = {"Bob", 183, 28}; // using starting values
    candidate fred = bob;            // using other object
    fred.name = "Fred";

    return 0;
}
```

<code>std::ostream</code>	Datentyp für <b>Output-Streams</b>
<p>Erfordert: <code>#include &lt;ostream&gt;</code> oder <code>#include &lt;iostream&gt;</code></p> <p>Beispielsweise <code>std::cout</code> hat den Typ <code>std::ostream</code>. Objekte des Typs <code>std::stringstream</code> können auch als <code>std::ostream</code> verwendet werden.</p> <p>Objekte des Typs <code>std::ostream</code> können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.</p>	
<pre>// POST: wrote the highscore of a given player to out. void print (std::ostream&amp; out, std::string name, int score) {     out &lt;&lt; "Player: " &lt;&lt; name &lt;&lt; " Score: " &lt;&lt; score &lt;&lt; "\n"; }  int main () {     print(std::cout, "Pete", 335);     print(std::cout, "Paula", 410);     return 0; }</pre>	

## Operatoren

operator...	Einen Operator überladen.
<p><a href="#">Operator-Überladung</a> wird zum Beispiel verwendet, um Operatoren (+, -, *, etc.) auf Structs zu definieren.</p>	
<pre>struct rational {     int n;     int d; // INV: d != 0 };  // POST: return value is the sum of a and b rational operator+ (const rational a, const rational b) {     rational result;     result.n = a.n * b.d + a.d * b.n;     result.d = a.d * b.d;     return result; }  // POST: return value is the sum of a and b rational operator+ (const rational a, const int b) {     rational result;     result.n = a.n + a.d * b;     result.d = a.d;     return result; }  int main () {     rational r = {1, 2};     rational s = {3, 4};     rational t = r + s; // first overload     std::cout &lt;&lt; t.n &lt;&lt; "/" &lt;&lt; t.d &lt;&lt; "\n"; // Output: 10/8     rational u = r + 3; // second overload     std::cout &lt;&lt; u.n &lt;&lt; "/" &lt;&lt; u.d &lt;&lt; "\n"; // Output: 7/2     return 0; }</pre>	