

## Datentypen

<code>struct</code>	Container für Datentypen
<p>Wichtige Befehle:</p> <p><b>Definition:</b> <pre>struct str_name {     int mem1;     bool mem2;     int mem3; };</pre></p> <p><b>Objekt erstellen:</b> <pre>str_name obj1;</pre></p> <p><b>mit Startwerten:</b> <pre>str_name obj2 = {3, true, 4};</pre></p> <p><b>aus anderem Objekt:</b> <pre>str_name obj3 = obj2;</pre></p> <p><b>Zugriff auf Member:</b> <pre>obj1.mem1</pre></p> <p>Die <i>Definition</i> eines Structs hat ein <code>;</code> am Schluss.</p> <p>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>).</p> <p>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>.</p>	
<pre>struct candidate {     std::string name;    // Name of the participant     int age;            // Her/his age };  int main () {     // Initialization     candidate mary;    // default-initialisation     std::cout &lt;&lt; mary.age;    // Undefined behavior     mary.name = "Mary"; mary.age = 43;     std::cout &lt;&lt; mary.age;    // Problem gone: mary.age is 43     candidate bob = {"Bob", 28}; // using starting values     candidate fred = bob;    // using other object     fred.name = "Fred";     return 0; }</pre>	

<code>class</code>	Datencontainer mit <b>Kapselung</b>
<p>Eine Klasse besteht aus Daten und Funktionen, genannt Member, und erlaubt deren Kapselung via Zugriffskontrolle: Auf Member im privaten Teil (<code>private</code>) einer Klasse kann nur durch die Klasse selbst, d.h., deren Member-Funktionen zugegriffen werden.</p> <p>Zugriff von ausserhalb der Klasse muss über öffentliche (<code>public</code>) Member erfolgen. Per default sind die Member einer Klasse privat.</p> <p>Einziger Unterschied gegenüber <code>structs</code>: Member in <code>structs</code> sind per default öffentlich (<code>public</code>).</p> <p>Deklarationsreihenfolge von Membern ist irrelevant.</p>	
<pre>class my_class { public: // public section     double some_public_member; private: // private section     double some_private_member; };  ... my_class inst; inst.some_public_member = 1.0; inst.some_private_member = 0.0; // ERROR: cannot access private                                 //          members directly</pre>	

Memberfunktion	Funktionalität auf Klassen
<p><b>Memberfunktionen</b> stellen Funktionalität auf einer Klasse bereit. Sie ermöglichen den kontrollierten Zugang zu den privaten Daten und privaten <b>Memberfunktionen</b>. Die <i>Deklaration</i> einer <b>Memberfunktion</b> erfolgt immer in der Klassendefinition, die <i>Definition</i> der Memberfunktion ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings die Zugehörigkeit zur Klasse explizit erwähnt werden mittels der <b>::-Schreibweise</b>.</p> <p>Der Aufruf einer Memberfunktion ist <code>obj.mem_func(arg1, arg2, ..., argN)</code>. Der Teil <code>obj.</code> kann weggelassen werden, falls <b>aus der Class heraus</b> auf einen Member des aufrufenden Objekts (siehe Eintrag <b>*this</b>) zugegriffen wird.</p>	
<pre>// Internal Definition vs. External Definition class Insurance { public:     void set_rate_i (const double v) { rate = v; }           // int.     void set_rate_e (const double v);     ... private:     double rate;     ... };  void Insurance::set_rate_e (const double v) {rate = v;} // ext. ----- // Call from Inside vs. Call from Outside class Insurance { public:     double get_rate () {         if (!is_up_to_date) update_rate();           // from inside         return rate;     }     double get_cost () {return get_rate() * ...;} // from inside     ... // e.g. stuff which sets the data members private:     bool is_up_to_date;     double rate;     double update_rate () { rate = ...; } };  ... Insurance insurance; ... std::cout &lt;&lt; insurance.get_rate();                 // from outside</pre>	

# Programmier-Befehle - Woche 10

<code>const</code> Memberfunktion	Unverändernde Memberfunktion
Das <code>const</code> bezieht sich auf <code>*this</code> . Es verspricht, dass durch die Funktionsausführung das implizite Argument nicht im Wert verändert wird.	
<pre>class Insurance { public:     double get_value() const {         return value; // same: return (*this).value;     }     ... // e.g. members which set the data members private:     double value; };</pre>	

Konstruktor	Datencontainer Initialisierung
<p><b>Konstruktoren</b> sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen. Sie werden bei der Variablendeklaration aufgerufen.</p> <p>Sie werden analog zu Funktionen überladen und bei der Variablendeklaration wie eine Funktion aufgerufen. Damit das funktioniert, muss der <b>Konstruktor</b> öffentlich (<code>public</code>) sein.</p> <p>Spezielle <b>Konstruktoren</b> sind der <b>Default-Konstruktor</b> (kein Argument), welcher automatisch erzeugt wird, falls eine Klasse keinen <b>Konstruktor</b> definiert, und der <b>Konversions-Konstruktor</b> (genau ein Argument), welcher die Definition <b>benutzerdefinierter Konversionen</b> ermöglicht.</p>	

( ... )

( ... )

```
class Insurance {
public:
    Insurance(double v, int r) // general constructor
        : value (v), rate (r) // initialize data members
        { update_rate(); }
    Insurance() // default constructor
        : value (0), rate (0) // initialize data members
        { }
    // other members
private:
    double value;
    double rate;
    void update_rate();
};
...
// General Constructor
Insurance i1 (10000, 10);
// default-Constructor, direct call
Insurance i3; // identical: Insurance i3 ();
...
-----
class Complex {
public:
    // Conversion Constructor (float --> Complex)
    Complex(const float i) : real (i), imag (0) { }
private:
    float real;
    float imag;
};
```

## Operatoren

<code>operator...</code>	Einen <b>Operator</b> überladen.
<p><b>Operator-Überladung</b> wird zum Beispiel verwendet, um Operatoren (+, -, *, etc.) auf eigenen Structs zu definieren.</p> <p>Mittels dem <code>operator...</code> Keyword ist es ebenfalls möglich, den Operator auszuführen. Das sollte man aber vermeiden, da damit der Code unlesbar wird.</p>	

( ... )

(...)

```
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 b_rat;
    b_rat.n = b; b_rat.d = 1; // b_rat is b/1
    return a + b_rat; // Use operator+ for two rationals (above)
}

int main () {
    rational r = {1, 2};
    rational s = {3, 4};
    rational t = r + s; // first overload
    std::cout << t.n << "/" << t.d << "\n"; // Output: 10/8
    rational u = r + 3; // second overload
    std::cout << u.n << "/" << u.d << "\n"; // Output: 7/2
    return 0;
}
```