Datentypen

`struct`	**Container** für Datentypen

Wichtige Befehle:

Definition:
```
struct str_name {
    int mem1;
    bool mem2;
    int mem3;
};
```
Objekt erstellen: `str_name obj1;`
mit Startwerten: `str_name obj2 = {3, true, 4};`
aus anderem Objekt: `str_name obj3 = obj2;`
Zugriff auf Member: `obj1.mem1`

Die *Definition* eines Structs hat ein `;` am Schluss.

Nur der Zuweisungsoperator (`=`) wird automatisch erstellt (und kopiert dann die Member einzeln). Die anderen Operatoren (z.B. `==`, `!=`, ...) muss man selbst passend überladen (siehe Eintrag `operator...`).

Bei der Default-Initialisierung eines Objekts des Typs `str_name` werden alle Member einzeln default-initialisiert. Für fundamentale Typen (`int`, `float`, usw.) bedeutet das, dass sie *uninitialisiert* sind, bis man ihnen nachträglich einen Wert zuweist. Das führt zu Problemen, falls man ihren Wert vorher schon ausliest.

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

int main () {
    // Initialization
    candidate mary;              // default-initialisation
    std::cout << mary.age;       // Undefined behavior
    mary.name = "Mary"; mary.age = 43;
    std::cout << 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;
}
```

`class`	Datencontainer mit **Kapselung**

Eine Klasse besteht aus Daten und Funktionen, genannt Member, und erlaubt deren Kapselung via Zugriffkontrolle: Auf Member im privaten Teil (`private`) einer Klasse kann nur durch die Klasse selbst, d.h., deren Member-Funktionen zugegriffen werden.

Zugriff von ausserhalb der Klasse muss über öffentliche (`public`) Member erfolgen. Per default sind die Member einer Klasse privat.

Einziger Unterschied gegenüber `structs`: Member in `structs` sind per default öffentlich (`public`).

Deklarationsreihenfolge von Membern ist irrelevant.

```
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
```

Memberfunktion	**Funktionalität auf Klassen**

Memberfunktionen stellen Funktionalität auf einer Klasse bereit. Sie ermöglichen den kontrollierten Zugang zu den privaten Daten und privaten Memberfunktionen. Die *Deklaration* einer Memberfunktion erfolgt immer in der Klassendefinition, die *Definition* der Memberfunktion ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings die Zugehörigkeit zur Klasse explizit erwähnt werden mittels der `::`-Schreibweise.

Der Aufruf einer Memberfunktion ist `obj.mem_func(arg1, arg2, ..., argN)`. Der Teil `obj.` kann weggelassen werden, falls aus der Class heraus auf einen Member des aufrufenden Objekts (siehe Eintrag `*this`) zugegriffen wird.

```
// 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 << insurance.get_rate();               // from outside
```

`const Memberfunktion`	**Unverändernde Memberfunktion**

Das const bezieht sich auf *this. Es verspricht, dass durch die Funktionsausführung das implizite Argument nicht im Wert verändert wird.

```
class Insurance {
public:
    double get_value() const {
        return value; // same: return (*this).value;
    }
    ... // e.g. members which set the data members
private:
    double value;
};
```

`Konstruktor`	Datencontainer **Initialisierung**

Konstruktoren sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen. Sie werden bei der Variablendeklaration aufgerufen.

Sie werden analog zu Funktionen überladen und bei der Variablendeklaration wie eine Funktion aufgerufen. Damit das funktioniert, muss der Konstruktor öffentlich (`public`) sein.

Spezielle Konstruktoren sind der Default-Konstruktor (kein Argument), welcher automatisch erzeugt wird, falls eine Klasse keinen Konstuktor definiert, und der Konversions-Konstruktor (genau ein Argument), welcher die Definition benutzerdefinierter Konversionen ermöglicht.

(...)

(...)

```
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

`operator...`	Einen **Operator überladen**.

Operator-Überladung wird zum Beispiel verwendet, um Operatoren (`+`, `-`, `*`, etc.) auf eigenen Structs zu definieren.

Mittels dem `operator...` Keyword ist es ebenfalls möglich, den Operator auszuführen. Das sollte man aber vermeiden, da damit der Code unlesbar wird.

(...)

(...)

```
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;
}
```