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...`).

Als Struct-Member können Arrays kopiert werden. Sie werden standardmässig eintragsweise kopiert.

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;
}
```

`std::ostream`	Datentyp für **Output-Streams**

Erfordert: `#include<ostream>` oder `#include <iostream>`

Beispielsweise `std::cout` hat den Typ `std::ostream`. Objekte des Typs `std::stringstream` können auch als `std::ostream` verwendet werden.

Objekte des Typs `std::ostream` können nicht direkt kopiert werden. Deshalb sollte man sie immer via Call-by-Reference an Funktionen übergeben.

```
// POST: wrote the highscore of a given player to out.
void print (std::ostream& out, std::string name, int score) {
    out << "Player: " << name << "  Score: " << score << "\n";
}

int main () {
    print(std::cout, "Pete", 335);
    print(std::cout, "Paula", 410);
    return 0;
}
```

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;
}
```