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
```

(Anstatt `int` und `bool` können die Member beliebige Typen haben.)

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 (Details, siehe Eintrag operator...). Den Zuweisungsoperator (`=`) kann man ebenfalls überladen, falls kein mitgliedsweises Kopieren gewünscht ist.

Arrays als Struct-Member 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 ihr Wert vorher schon ausgelesen wird.

(...)

(...)

```
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;
}
```

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: Outputs the highscore of a player to the terminal.
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;
}
```

`const` Referenzen	`const`-**Alias** für bestehende Variable

Im Prinzip funktionieren `const` Referenzen so wie normale Referenzen, bloss dass der Schreibzugriff auf das Ziel der Referenz *via diese Referenz* verboten ist.

Ein weiterer Unterschied ist, dass `const` Referenzen R-Werte beinhalten können. Dann wird jeweils ein temporärer Speicher für den R-Wert erstellt, der solange gültig ist, wie die `const` Referenz selbst. Dies erlaubt beispielsweise, eine Funktion bezüglich Call-by-Reference trotzdem mit R-Werten aufzurufen.

Zu beachten ist auch, dass man keine nicht-`const` Referenz mit einer `const` Referenz initialisieren darf.

```
double a = 3.0;
double& b = a; // non-const reference
const double& c = a; // const reference

c = 4.0; // Error: write-access forbidden
a = 5.0; // this works, a can be changed through itself
b = 6.0; // this works, a can be changed through non-const refs

std::cout << c << "\n"; // Output: 6.0, read-access is allowed.
double& d = c; // Error: non-const ref from const ref not allowed
const double& e = 5.0; // this works for const references.
```

Operatoren

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

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

Die Operator-Überladung funktioniert auch für Operatoren, welche als Member eines Structs (oder Class → siehe später) definiert sind. Operatoren als Member werden aber erst später in der Vorlesung behandelt.

Mittels `operator...` Keyword ist es ebenfalls möglich den Operator auszuführen. Das sollte man aber vermeiden, da damit der Code unleserlich 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 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 << t.n << "/" << t.d << "\n";
    rational u = r + 3; // second overload
    std::cout << u.n << "/" << u.d << "\n";
    return 0;
}
```