24. November

struct: Ein struct ist ein "selbstgebauter Typ".

Form:

```
struct name {
    // Member-Variabeln
};
```

Ein Beispiel wäre

```
struct student {
    int alter;
    std::string name;
    std::vector<double> noten;
};
```

Initialisierung: `student harry = {19, "harry", {1, 1.1, 1.05}}`

Abrufen: `harry.alter`

Function Overloading: Wir können mehrere Funktionen mit gleichem Namen haben, aber anderem Eingabetyp. Beispiel:

```
int foo (int a) {
    return a;
}
int foo (double a) {
    return 1;
}
```

Wie wir sehen, haben wir nun zwei Funktionen mit gleichem return-Typ und gleichem Namen. Die Funktion wird also aufgerufen, welche dem Typen entspricht, welches eingegeben wird.

Nützlich sind function overloads vor allem bei operatoren. So können wir eine Addition, oder ein && für einen bestimmten struct oder eine bestimmte class (siehe weiter unten) definieren.

nächste Seite =>

KLASSEN

Ähnlich zu Structs, aber viel mächtiges !

private, public: anders als bei den structs können Klassen einen "private" und eine "public" Bereich haben. Alles was wir im private-Bereich definiesen, kann nur in der Definition der Klasse selber aufgerufen werden. Ein struct hingegen hat nur einen public Bereich

Memberfunktionen: sind Funktionen in Klassen. Sie können also auf die privaten Daten der Klasse zugreifen und ermöglichen so den Zugang zu ihnen von aussen.

Bsp.

```
class Student {
    public:
        double get_mark() const {
            return mark;
        }
    private:
        double mark;
};
```

Memberfunktion

Der Aufruf erfolgt dann mit

```
cout << harry.get_mark();
```

Das const bei der Funktion bezieht sich auf harry (oder auch *this (siehe später))

Konstruktoren: Memberfunktionen mit dem Namen der Klasse. Sie werden bei der Variablendeklaration aufgerufen.

Bsp.

```
class Student {
    public:
        Student (double n) {
            mark = n;
        }
        double get_mark () const {
            return mark;
        }
    private:
        double mark;
};
```

Constructor

oder auch kürzer

```
Student (double n) : mark (n) {}
```

im int main () sieht das dann so aus:

```
Student harry (2.1);
```

Falls wir einen Student ohne Wert initialisieren möchten, können wir einen Default-Konstruktor definieren (ohne Argument)

```
Student ();
```