Datentypen

`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 = Insurance();
...
-----------------------------------------------------------------------
class Complex {
public:
    // Conversion Constructor (float --> Complex)
    Complex(const float i) : real (i), imag (0) { }
private:
    float real;
    float imag;
};
```

Iteratoren

Iterator (auf Vektor)	**Iterieren** über einen Vektor.

Erfordert: `#include<vector>`

Wichtige Befehle (gelte `std::vector<int> a (6, 0);`):

Definition:	`std::vector<int>::iterator itr = ...;`
Iterator auf `a[0]`:	`a.begin()`
Past-the-End-Iterator:	`a.end()`
Zugriff auf Iterator:	`itr = otr_itr // Iterator gets new target.`
Zugriff auf Target:	`*itr = 5 // Target gets new value 5.`
Vergleich:	`itr == otr_itr // Same target?`
	`itr != otr_itr // Different targets?`

Anstelle des `...` in der Definition eines Iterators müssen andere Iteratoren stehen (z.B. `a.begin()`).

Um lange Zeilen zu vermeiden, siehe Eintrag Typ-Alias.

Der `*` Operator, um auf das Targets eines Iterators zuzugreifen, wird auch Dereferenz-Operator genannt.

```
// Example for vectors.

// Read 6 values into a vector
std::cout << "Enter 6 numbers:\n";
std::vector<int> a (6, 0);
for (std::vector<int>::iterator i = a.begin(); i < a.end(); ++i)
    std::cin >> *i; // read into object of iterator

// Output:   a[0]+a[3],  a[1]+a[4],  a[2]+a[5]
for (std::vector<int>::iterator i = a.begin(); i < a.begin()+3; ++i) {
    assert(i+3 < a.end()); // Assert that i+3 stays inside.
    std::cout << (*i + *(i+3)) << ",  ";
}
```

`const (Iterator)`	**kein Schreibzugriff** auf das Objekt

Vorsicht: Einen `const`-Iterator erzeugt man mittels
`std::vector<int>::const_iterator ...`
und **nicht** mittels
`const std::vector<int>::iterator ...`

Die zweite Version erzeugt einen Iterator, den man nicht herumschieben kann. In dieser Vorlesung gehen wir aber nur auf die Iteratoren näher ein, welche den Schreibzugriff auf *das Objekt* verbieten (erste Variante oben).

```
std::vector<int> a (6, -8); // a is: -8 -8 -8 -8 -8 -8

std::vector<int>::const_iterator itr = a.begin() + 3;
*itr = 4;        // NOT valid
itr = a.begin(); // valid (itr now points to a[0])
```

Bereichsbasierte for-Schleife	

Sequenzielle Iteration mittels eines Iterators über einen `std::vector<int>` (const-Iterator möglich; andere Container möglich):

```
std::vector<int> v(3); // v == 0, 0, 0
for (std::vector<int>::iterator it = v.begin(); it != v.end();
++it) {
   std::cout << *it; // 000
}
```

Kann alternativ auch wie folgt geschrieben werden:

```
for (int i : v) std::cout << i; // 000
```

Wird dann zu Iterator-basierter Schleife übersetzt.

Modifizierender Zugriff ist auch möglich:

```
for (int& i : v) i += 3;
for (int i : v) std::cout << i; // 333
```

Datentypen

`set`	Datentyp für **Mengen** (jedes Element kommt nur einmal vor).

Erfordert: `#include<set>`

Wichtige Befehle (Sei `b = some_vec.begin(); e = some_vec.end();`):

Definition: `std::set<int> my_set (b, e);`
(Initialisiert `my_set` mit den Werten im Bereich `[b,e)`.)

Die **Iteratoren** der `set`s funktionieren wie die Iteratoren der Vektoren, **aber:**

Keine: `[], +, -, <, >, <=, >=, +=, -=`
Zum **Verschieben** nur: `++..., ...++, --..., ...--, =`
Zum **Vergleichen** nur: `==, !=`

```
// Determine All Occurring Numbers
std::cout << "Enter 100 numbers:\n";
std::vector<int> nbrs (100);
for (int i = 0; i < 100; ++i)
    std::cin >> nbrs[i];

std::set<int> uniques (nbrs.begin(), nbrs.end());

// Output
using Sit = std::set<int>::iterator;
for (Sit i = uniques.begin(); i != uniques.end(); ++i)
    std::cout << *i << "  ";

// This does not work:
for (int i = 0; i < uniques.end() - uniques.begin(); ++i)
    std::cout << uniques[i];
```

Standard-Funktionen

`std::fill(b, p, val)`	Wert `val` in einen **Bereich** `[b,p)` **einlesen**
Erfordert: `#include<algorithm>`	

```
// Goal: Generate vector: 4 4 4 2 2
std::vector<int> vec (5, 4);              // vec: 4 4 4 4 4
std::fill(vec.begin()+3, vec.end(), 2);   // vec: 4 4 4 2 2
```

`std::find(b, p, val)`	`val` **suchen** im Bereich `[b,p)`

Erfordert: `#include<algorithm>`

Zurückgegeben wird ein **Iterator** auf das *erste* gefundene Vorkommnis.

Wenn `std::find` nicht fündig wird, gibt es den Past-the-End-Iterator `p` zurück. (Beachte: Past-the-End ist bezüglich Bereich `[b,p)` gemeint.)

```
using Vit = std::vector<int>::iterator;
std::vector<int> vec (5, 2);
vec[3] = -7

// Goal: Find index of -7 in vec: 2 2 2 -7 2
Vit pos_itr = std::find(vec.begin(), vec.end(), -7);
std::cout << (pos_itr - vec.begin()) << "\n"; // Output: 3
```

`std::sort(b, e)`	Bereich `[b, e)` **sortieren**

Erfordert: `#include<algorithm>`

`std::sort` funktioniert nur, wenn Random-Access Iteratoren für `b` und `e` übergeben werden. Somit funktioniert `std::sort` z.B. für Felder und Vektoren, aber nicht z.B. für Sets.

(...)

(...)

```
std::vector<int> vec = {8, 1, 0, -7, 7};
std::sort(vec.begin(), vec.end()); // vec:  -7 0 1 7 8
```

`std::min_element(b, p)`	**Iterator auf Minimum** im Bereich `[b,p)`

Erfordert: `#include<algorithm>`

Wenn das Minimum nicht eindeutig ist, so wird ein Iterator auf das erste Vorkommnis zurückgegeben.

```
// Goal: Make sure that all inputs are > 0
std::vector<int> vec (10, 0);
for (int i = 0; i < 10; ++i)
    std::cin >> vec[i];

assert( *std::min_element(vec.begin(), vec.end()) > 0 );
    // Note: We have to dereference the (r-value-)iterator.
```