Zu Beachten

In diesem Summary werden bei Member-Funktionen die PRE- und POST-Conditions aus Platzgründen oftmals weggelassen. Bei eigenen Programmen müssen diese aber unbedingt mit angegeben werden.

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 zugeriffen 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 Insurance {        // Definition
public: // public section
    double get_rate();
    std::string remark; // public data-member
                        //   (possible, but not recommended)
private: // private section
    bool is_up_to_date_;  // data-member
    double rate_;         // data-member
    void update_rate_();  // member-function
};

int main() {
    Insurance insurance;

    // Accessing public data member analoguous to structs:
    insurance.remark = "A remark";
    std::cout << insurance.remark;
    return 0;
}
```

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 Klassendefiniton, die *Definition* ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings die Zugehörigkeit zur Class explizit erwähnt werden mittels der `::`-Schreibweise.

Der Aufruf einer Memberfunktion ist: `<expr>.my_func(arg1, arg2, ..., argN);`, wobei `<expr>` ein Ausdruck vom Typ Klasse ist (z.B. der Name eines Objekts). Der Teil `expr.` kann weggelassen werden, falls ein Aufruf auf einen Member des aufrufenden Objekts (siehe Eintrag `*this`) erfolgt.

```
// 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
```

*this	Zugriff auf **implizites Argument**

Memberfunktionen einer Klasse haben ein implizites Argument vom Typ der Klasse. Dieses implizite Argument ist eine Referenz auf das aufrufende Objekt. Das Schlüsselwort *this ermöglicht den Zugriff.

Bei Zugriffen von innerhalb einer Klasse aus auf Daten-Member oder Member-Funktionen wird das implizite Argument automatisch verwendet. Man muss es dann also nicht unbedingt explizit angeben (siehe Eintrag Memberfunktion). *this kann aber auch explizit angegeben werden, falls z.B. die Verkettung von Funktionsaufrufen ermöglicht werden soll (vor allem im Zusammenhang mit Operatoren als Memberfunktionen relevant, wenn eine Referenz auf den linken Operanden zurückgegeben wird).

```
// General example
class Human {
public:
    void set (const int a) { age_ = a; } // or (*this).age_ = a;
    void print1 () const { std::cout << (*this).age_; }
    void print2 () const { std::cout << age_; } // equivalent
private:
    int age_;
};
...
Human me;  me.set(175);
me.print1();  // 175
me.print2();  // 175
-----------------------------------------------------------------
// Chaining Example
class Complex {
public:
    // Note: In most applications a reference should be returned
    Complex& operator+=(const Complex& b) {
        real_ += b.real_;
        imag_ += b.imag_;
        return *this;
    }
    ... // other members
private:
    float real_;
    float imag_;
};
```

`const Memberfunktion`	**Unverändernde Memberfunktion**

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

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

`typedef-Member`	Kapseln von **Typendefinitionen**

Klassen erlauben die Kapselung von Typendefinitionen. Dies ermöglicht eine (kompatible) Redefinition dieser Typen zu einem späteren Zeitpunkt, ohne dass die User der Klasse dann ihre Projekte umschreiben müssen. (Die Änderung erfolgt an einer einzigen Stelle.)

Bei Verwendungen ausserhalb der Klasse muss mit der ::-Schreibweise auf den typedef-Member zugegriffen werden.

```
class My_Class {
public:
    typedef double My_Double;
    My_Double d;
};

int main() {
    My_Class::My_Double d = 0;
    d += 1; // can be used exactly like double
    return 0;
}
```

Konstruktor	Datencontainer **Initialisierung**

Konstruktoren sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen und 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); // direct call
Insurance i2 = Insurance(10000,10); // indirect: copy
// 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_;
};
```

Dynamische Datentypen

`new`, `delete`	**Objekt mit dynamischer Lebensdauer** erstellen.

Mit `new` wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener Constructor aufgerufen wird. Bei `delete` wird zuerst ein Destruktor aufgerufen, bevor der Speicherplatz freigegeben wird.

Der Rückgabewert von `new` ist ein *Pointer* auf das neu erstellte Objekt. Wird mit `delete` ein Objekt gelöscht, so sollte man immer *alle* Pointer, die auf das Objekt zeigen, auf `0` setzen.

Jedes `new` braucht ein `delete`. Sonst existieren die erstellten Objekte bis zum Ende des Programms, was je nach Laufdauer eine grosse Speicherverschwendung ist.

```
Class My_Class {
public:
    My_Class (const int i) : y_(i) { std::cout << "Hello"; }
    int get_y () { return y_; }
private:
    int y_;
};

...
My_Class* ptr = new My_Class (3); // outputs Hello
My_Class* ptr2 = ptr; // another pointer to the new object
std::cout << (*ptr).get_y(); // Output: 3
delete ptr;
ptr = 0;
ptr2 = 0; // has to be done !separately!
...
```

`new, delete[]`	**Ranges mit dynamischer Lebensdauer und Länge** erstellen.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
delete range; // ERROR: must say: delete[]
delete[] range; // This works
```

Copy-Constructor	**Kopier-Initialisierung**

Der Copy-Constructor ist der Constructor, dessen Argumenttyp `const My_Class&` ist.

```
struct Customer {
    std::string name;
    int duration;
    int amount_insured;
};

class Insurance {
public:
    Insurance (const Insurance& rhs)
        : length_(rhs.length_), ... // copy remaining data mbrs
        {
            cust_ = new Customer [length_];
            for (int i = 0; i < length_; ++i)
                cust_[i] = rhs.cust_[i];
        }
    ... // other public members
private:
    Customer* cust_; // pointer to an array containing customers
    int length_;     // length of cust_
    ... // other private members
};
```

operator=	**Kopie-Zuweisung**

Eng verwandt mit `operator=` ist der Copy-Constructor. Der Unterschied ist, dass der Copy-Constructor nur bei der Initialisierung aufgerufen wird, `operator=` hingegen nur *nach* der Initialisierung. z.B.

```
My_Class a (5, 6), c (4, 4); // Call a general constructor
My_Class b = a;              // Call copy-constructor
c = b;                       // Call operator=
```

Der Copy-Constructor kann in der Tat anders als `operator=` (für rechte Seiten des selben Typs) implementiert werden müssen. Ein Beispiel hierfür sind Klassen, welche dynamisch generierte Member haben (genauer: Pointer, die auf solche zeigen). Dann muss bei `operator=` in vielen Fällen zuerst der aktuell vorhandene Member gelöscht werden, bevor die Kopie erstellt werden kann. Dies ist beispielsweise beim Stack aus der Vorlesung relevant.

`operator=` gibt im Normalfall eine Referenz auf seinen linken Operanden zurück.

Faustregel: Meistens führt `operator=` zuerst die Aufgaben des Destructors, und dann die Aufgaben des Copy-Constructors aus.

```
// for Customer-struct see example on Copy-Constructor

class Insurance {
public:
    Insurance& operator= (const Insurance& rhs) {
        delete[] cust_; // delete current customers
        cust_ = new Customer [length_];
        for (int i = 0; i < length_; ++i)
            cust_[i] = rhs.cust_[i];
        length_ = rhs.length_;
        ... // copy other data members
        return *this; // return a reference to left operand
    }
    ...
};
```

Destruktor	**Class abbauen**

```
// for Customer-struct see example on Copy-Constructor

class Insurance {
public:
    ~Insurance () { delete[] cust_; } // free dynamic space
    ...
};
```