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 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
```

`*this`	Zugriff auf **implizites Argument**

Memberfunktionen einer Klasse haben ein implizites Argument, nämlich das aufrufende Objekt. Und `this` ist ein Zeiger darauf. Via `*this` kann man darauf zugreifen.

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`). Man muss `*this` aber mindestens explizit verwenden, falls z.B. eine Referenz auf das implizite Argument zurückgegeben werden soll.

```
// 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
-----------------------------------------------------------------------
// Another 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 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;
};
```

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. 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 ();
...
-----------------------------------------------------------------
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 Konstruktor 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-Konstruktor	**Kopier-Initialisierung**

Der Copy-Konstruktor ist der Konstruktor, 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=	**Kopier-Zuweisung**

Eng verwandt mit `operator=` ist der Copy-Konstruktor. Der Unterschied ist, dass der Copy-Konstruktor 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=
```

`operator=` kann anders als der Copy-Konstruktor implementiert werden müssen. Ein Beispiel sind Klassen, welche Pointer auf dynamisch generierte Objekte als Member haben. Dann muss bei `operator=` meistens zuerst das aktuell vorhandene Objekt 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-Konstruktors aus.

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

class Insurance {
public:
    Insurance& operator= (const Insurance& rhs) {
        // Cleanup of current customers
        delete[] cust;

        // Copy over the customers from rhs
        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
    }
    ... // other members
};
```

Destruktor	**Class abbauen**

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

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