Member-Operator

❑ Beispiel:

```
class Complex {     double real, imag;
public:             Complex (double r=0, double i=0);
                    Complex operator + (const Complex& c) const;
};

Complex Complex :: operator + (const Complex& c) const {
    Complex result (real+c.real, imag+c.imag);
    return result;
}
...

Complex c1, c2, c3;
c1 = c2 + c3;                   // “normaler” Aufruf
c1 = c2 . operator + (c3)   // Daraus vom Compiler generierter Aufruf
```


cast-Operatoren

❑ Operatoren zur impliziten Typumwandlung definierbar

❑ kein Returndatentyp, ist implizit Operatorname

❑ Beispiel für **int**:

```
class MyInteger {
            int value;
public:     MyInteger (int val=0): value (val) { }
            operator int ( ) { return value; }
};
...
MyInteger x, y;
x = 2;
y = x + 3;
```


Zuweisung, Copy

❑ Zuweisung: immer an ein existierendes Objekt

```
Person hans;
hans = peter;
```

- **Person & Person::operator= (Person&)**

❑ Copy: immer in Verbindung mit Deklaration

```
void TestFunc (Person p) { ... }
void main ( ) {
    Person hans = peter;
    TestFunc (hans);
}
```

- **Person::Person (const Person &)**

Index-Operator [ ]

❑ Operator **[ ]** ist ein binärer Operator

❑ sollte eine Referenz zurückliefern

❑ Beispiel:

```
class Vektor {
    double values [3];
public:
    double& operator [ ] (int);
};

double& Vektor::operator [ ] (int x) {return values [x-1];}

Vektor z;
z [3] = z [2] + 1.01234;
```

Funktions-Operator ( )

❏ Operator () ist ein Operator mit beliebig vielen Argumenten

❏ Beispiel:

```
class Matrix3x3 {
    double values [3] [3];
public:
    double& operator ( ) (int, int);
};

double& Matrix3x3 :: operator ( ) (int x, int y) {
    return values [x-1] [y-1];
}

Matrix3x3 z;
z (3, 2) = z (1, 2) + 1.01234;
```


Verschiedene Operatoren

❏ Beispiel für sinnvolle Operatorüberladungen:

```
class Complex {
    Complex(const Complex&);
    Complex(double);
    Complex& operator = (const Complex&);
    Complex& operator = (double);
    Complex operator - (const Complex&) const;
    Complex operator - (double) const;
    Complex operator - ( ) const;
    Complex& operator -= (const Complex&) const;
    Complex& operator -- ( );     // Präfix Operator --
    Complex operator -- (int);    // das ist der Postfix-Operator --, Par. ist dummy
    // operator double ( ) const; // nicht sinnvoll
};
```


Temporäre Objekte

❏ Erzeugung bei Parameterübergabe, cast, Rückgabe, explizit

❏ Haben keinen Namen

❏ Verschwinden nach der nächsten Anweisung

❏ Beispiel:

```
class MyInteger { ...
    public: MyInteger(int i);
};
...
MyInteger i = 2;        // das ist direkt der Konstruktor
i = 2;                  // alle 3 Varianten bewirken dasselbe
i = (MyInteger) 2;      // (erzeugen temp. Objekt)
i = MyInteger (2);
```


Exceptions

❏ Bisherige Fehlerbehandlung:

- Terminieren des Programms
- Ignorieren des Fehlers
- Aufrufen einer Behandlungsroutine
- Zurückliefern eines Fehlerstatus

Nachteil:

- Fehler muß über ges. Aufrufhierarchie zurückgegeben werden.
- Fehler im Konstruktor nicht behandelbar.
- Jede Funktion muß bei jedem Aufruf Fehler behandeln

Exceptions

- ❏ mit “try” wird die Ausführung eines Blockes versucht
- ❏ mit “throw” wird die Fehlerbehandlung aktiviert
- ❏ mit “catch” kann auf den Fehler reagiert werden
- ❏ Kontrollierte Fehlerbehandlung:
  - Reaktion auf ganz bestimmte Fehler möglich
  - Defaultbehandlung: Terminieren des Programms
  - Laufzeitbibliothek verwendet ebenfalls Fehlerbehandlung

Exceptions: Beispiel

```
class MathError {char* message;
                  public: MathError (char* m): message(m) { }
                         void Print ( ) { cout << message; }
};

int div (int x, int y) {
    if (y != 0) return x / y; else throw (MathError(“Div durch 0”));
}
void Calculate( ... ) { int x,y,z,t; x = div (y, z); t = div (x, z); }

try { ... Calculate( ... ); ... ; Calculate( ... ); ... }
catch (MathError m) {
    m.Print ( );
    throw (m);
}
```


Exceptions: Funktionsweise

- ❏ Exceptions werden bis zum passenden Exceptionhandler hochgereicht
- ❏ kein passender Handler -> Abbruch des Programms
- ❏ beim Hochreichen werden alle Destruktoren aufgerufen
- ❏ Es wird der erste typkonforme Exceptionhandler aufgerufen
- ❏ Durch Klassenhierarchie: Gruppierung von Fehlern

```
class MathException { };
class DivByZero: public MathException { };
class RangeOverflow: public MathException { };
...
try { ... Calculate( ); ... };
catch (RangeOverflow m) {....} // RangeOverflow behandeln
catch (MathException m) {....} // Alle anderen math. Fehler behandeln
```


Exceptions: Resourceproblem

- ❏ C-Beispiel:

```
{   ... open(“text.doc”);
    ... if (error) {close(...); return error;}
    ... close(...);
}
```

- ❏ C++ mit Exceptions: (“Resource aquisition is initialisation”)

```
class FileObject { ...     FileObject(...) { open (...); }
                           ~FileObject( ) { close (...); }
};
{   ... FileObject f (“text.doc”);
    ...            // Fehler werden automatisch durchgereicht
    ...            // close automatisch (~)
}
```