Themen

- Statische Methoden
- `inline` Methoden
- `const` Methoden
- `this` Zeiger
- Destruktor
- Kopierkonstruktor
- Überladen von Operatoren

Statische Methoden

- Klassenmethoden – Merkmal der Klasse nicht eines einzelnen Objekts
- Kann ausgeführt werden ohne vorher ein Objekt zu erzeugen
- Zum Deklarieren einer statischen Methode wird der Deklaration das Schlüsselwort **`static`** vorangestellt

statische Methode Beispiel

```
class Viereck
{
    ...
  public:
     static float laenge(
              float x1,float y1,
              float x2,float y2);
};
```

statische Methode Beispiel

```
float Viereck::laenge(
        float x1,float y1,
        float x2, float y2)
{
    float result = pow(x1-x2,2);
    result += pow(y1-y2,2);
    return sqrt(result);
}
```

statische Methode Beispiel

```
#include "Viereck.h"
int main()
{
   float x1=5.0f , y1=3.0f;
   float x2=6.0f , y2=17.0f;
   float laenge=Viereck::laenge(x1,y1,x2,y2);
   Viereck v;
   laenge=v.laenge(x1,y1,x2,y2);
}
```

Was passiert bei einem Funktionsaufruf?

1) Das Programm speichert die Argumente, die der Funktion übergeben werden und einige andere Daten auf dem Stackspeicher
2) Das Programm springt an den Beginn der Funktion
3) Die Funktion wird abgearbeitet
4) Das Programm stellt seinen alten Zustand wieder her, räumt den Stack auf und springt an die nächste Anweisung nach dem Funktionsaufruf

`inline` Methoden

- um eine Methode als **`inline`** zu deklarieren, wird der Methode das Schlüsselwort **`inline`** vorangestellt
- eine als **`inline`** deklarierte Methode ist ein **Hinweis** an den Compiler, dass er den Methodenaufruf durch den Funktionskörper ersetzen soll
- Inline Methoden müssen auch “inline” definiert werden, d.h. die Implementierung **muss** in der Header Datei stehen

`inline` Beispiel

```
// Bruch.h
class Bruch
{
    ...
    inline unsigned int getZaehler()
    {
      return zaehler;
    }

};
```

`const` Methoden

- **`const`** Methoden ändern das Objekt für das sie aufgerufen werden nicht
- um eine **nicht statische** Methode als **`const`** zu deklarieren stellt man das Schlüsselwort **`const`** hinter die Deklaration der Methode
- das Schlüsselwort **`const`** ist Teil der Signatur der Methode
  - es kann die gleiche Methode auch nochmal ohne **`const`** Schlüsselwort geben
  - bei der Implementierung muss das Schlüsselwort auch angegeben werden

const Beispiel

```
class Viereck
{
  public:
    float getUmfang();
};

const Viereck v(1,2,3,4,5,6,7,8);
float umfang=v.getUmfang(); // Fehler
```

const Beispiel

```
class Viereck
{
  public:
    float getUmfang() const;
};

const Viereck v(1,2,3,4,5,6,7,8);
float umfang=v.getUmfang(); // OK
```

`this` Zeiger

- Jede Klasse enthält einen **`this`**-Zeiger
- Zeigt auf das Objekt, von welchen eine Methode aufgerufen wurde
- nur verfügbar in nicht statischen Methoden
- `this` ist ein Zeiger auf das aktuelle Objekt, somit ist `*this` das Objekt selbst
- unveränderbar

this Zeiger - Beispiel

```
class Quadrat
{
   float seitenlaenge;
   public:
     float getUmfang();
}

float Quadrat::getUmfang()
{
   return 4 * this->seitenlaenge;
}
```

- this->: muss nicht explizit verwendet werden, wird implizit vom Compiler angenommen

this Zeiger - Beispiel

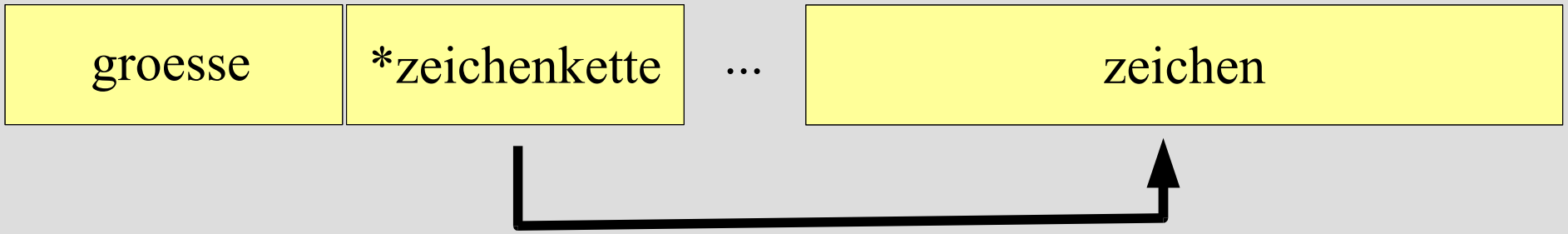
```
class Quadrat
{
    float seitenlaenge;
  public:
    float getUmfang();
    void setSeite(float seitenlaenge);
};
float Quadrat::getUmfang() {
   return 4 * this->seitenlaenge;
}

void Quadrat::setSeite(float seitenlaenge)
{
   this->seitenlaenge = seitenlaenge;
}
```

Destruktor Beispiel

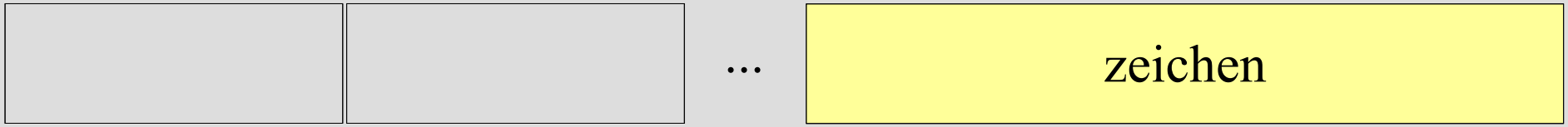
```
class String
{
  protected:
    int groesse;
    char *zeichenkette;
};

String::String(int g):groesse(g)
{
   zeichenkette=new char[groesse];
}
```

Destruktor Beispiel

```
String s(40);
```



Destruktor Beispiel



Destruktor

- Zur **Speicherfreigabe** von Objekten
- Wird automatisch **vor** Zerstörung eines Objekts aufgerufen – **Aufräumarbeiten**
- Nicht automatisch vom Compiler erzeugt!!!
- Name beginnt mit **Tilde** (~)

```
~Klassenname(){}
```

- **Keine** Parameter
- **Keine konstanten** Destruktoren

Destruktor-Beispiel

```
class String
{
   int groesse;
   char *zeichenkette;

   public:
      ~String(void);
};

String::~String()
{
     delete [] Zeichenkette;
}
```

Destruktor-Beispiel 2

- `delete` ruft Destruktor auf (wenn vorhanden)

```
struct NamenListe {
   Zeichen name;
   NamenListe *vorgaenger;
   ~NamenListe();
}

NameListe::~NameListe() {
   delete vorgaenger; // aktiviert Destruktor des Vorgängers
}
```

- delete arbeitet rekursiv !!! - ruft Vorgänger Destruktoren auf bis Null-Zeiger erreicht

Delete → Destruktor n → delete → Destruktor n–1 → ... → delete 0

Destruktor - Objekte

- Destruktor wird für Objekte nach folgenden Regeln aktiviert:
  - `Objekt` ist eine **globale Variable** – Destruktor am Programmende aktiviert
  - `Objekt` ist eine **lokale Variable** oder ein **Parameter** – Destruktor aktiviert, wenn Funktion oder Block beendet werden, in denen `Objekt` definiert worden ist
  - `Objekt` ist eine **anonyme Variable im Heap** (`*p` zeigt auf `Objekt`) - Wenn "`delete p`" aufgerufen wird, wird Destruktor von *p aktiviert

Zusammenfassung

- **Konstruktor**:
  - Konstruiert nicht, sondern initialisiert **nach** Erzeugung eines Objekts
- **Destruktor**:
  - Vernichtet nicht, sondern macht Aufräumarbeiten **vor** der Vernichtung eines Objekts
- **`new`**:
  - Erzeugt und ruft **danach** den Konstruktor auf
- **`delete`**:
  - Vernichtet und ruft **vorher** den Destruktor auf

Kopierkonstruktor (Copy)

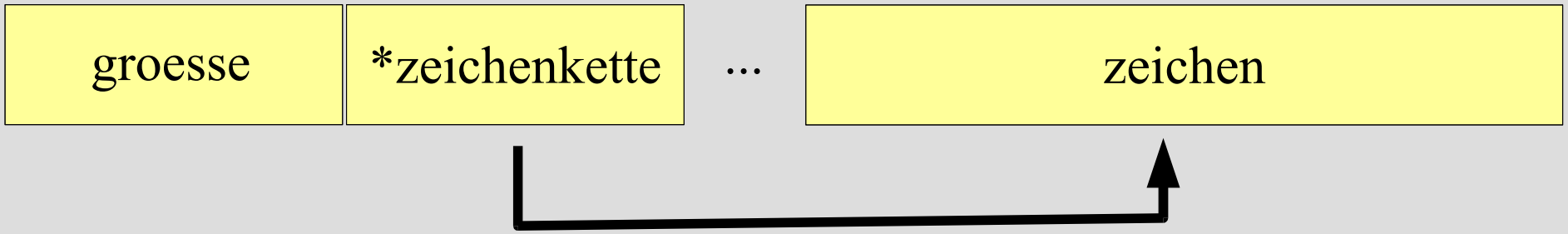
- Gibt eine Referenz auf ein Objekt derselben Klasse zurück (erstellt Kopie)

```
Klasse(const Klasse &Objektkopie)
```

- Um ein Objekt
  - mit einem Objekt derselben Klasse zu initialisieren
  - an Funktion zu übergeben
  - als Funktionswert zurückzugeben
- Falls **kein** Kopierkonstruktor vorhanden (automatisch vom Compiler erzeugt): jedem Attribut wird der Wert des zu kopierenden Objekts zugewiesen (Byte für Byte) -> **fehleranfällig (z.B. Zeiger)** !!!

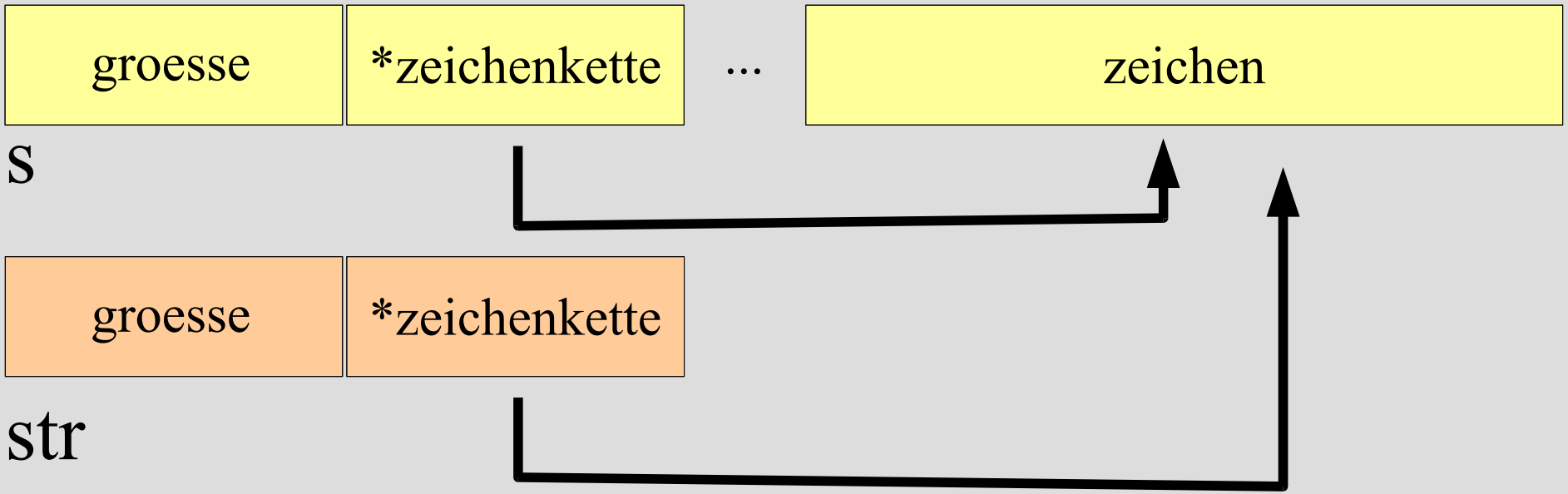
Kopierkonstruktor Beispiel

```
String s(40);
```



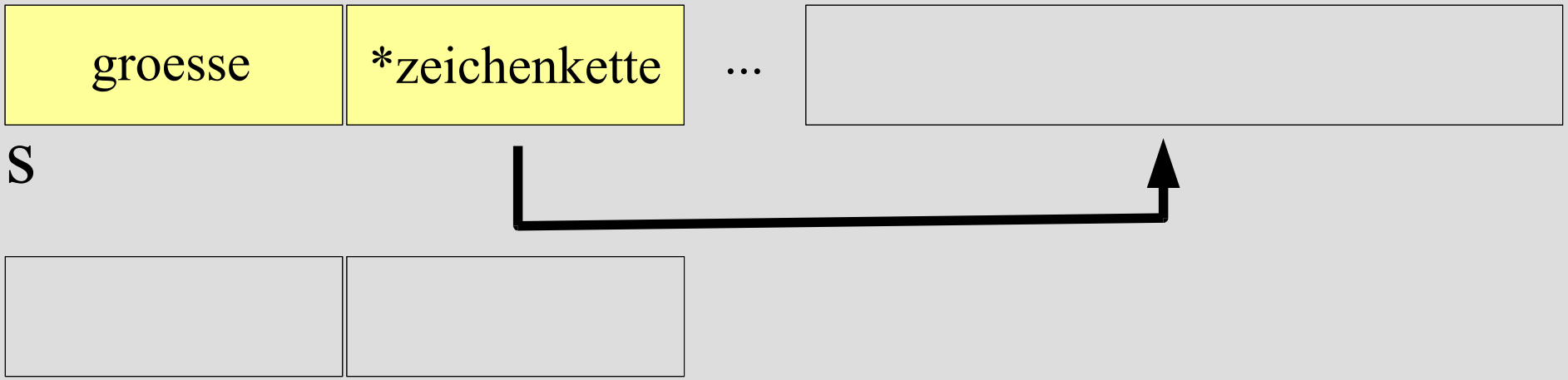
Kopierkonstruktor Beispiel

```
void printString(String str);

String s(40);
printString(s);
```



Kopierkonstruktor Beispiel

```
void printString(String str);

String s(40);
printString(s);
```

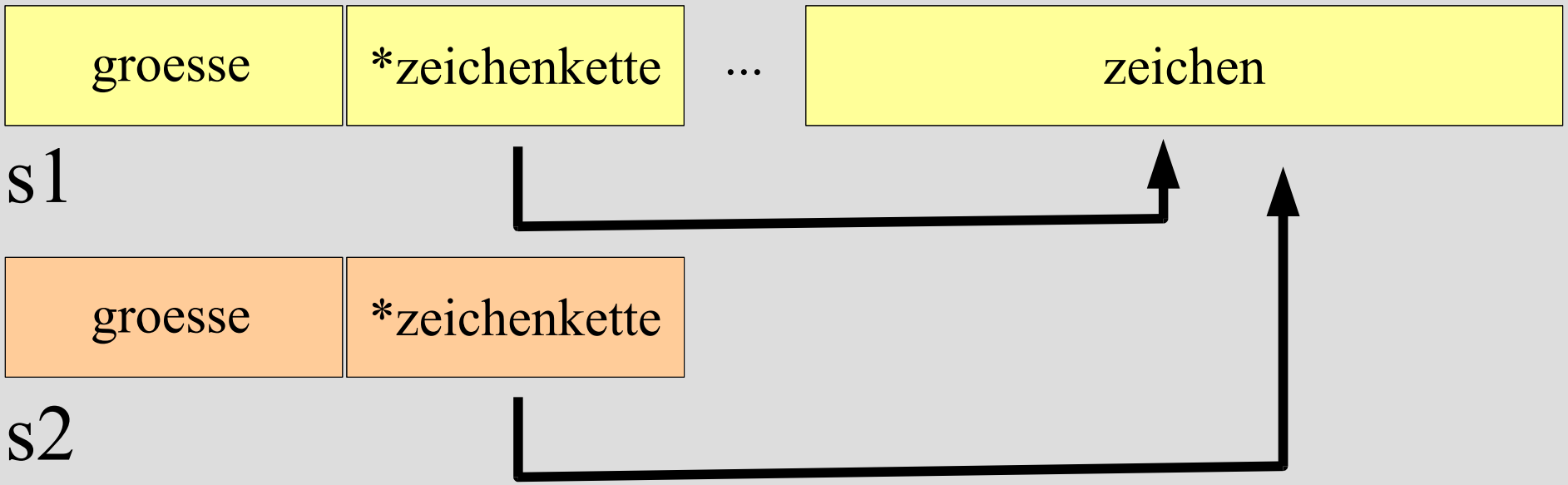


Kopierkonstruktor - Beispiel

```
class String
{
   int groesse;
   char *zeichenkette;

   public:
         ...
      String(const String& einString);
};

String::String(const String& einString)
{
   groesse = einString.groesse;
   zeichenkette = new char[groesse];
   strcpy(zeichenkette, einString.zeichenkette);
}
```

Zuweisungs Beispiel

```
String s1(40);
String s2 = s1;
```



Zuweisungsoperator überladen

- Memberfunktion für den zu überladenen Zuweisungsoperator definieren:

```
Typ& Typ::operator= (const DTYP& param)
```

- **Typ** = Klasse, für die der Zuweisungsoperator überladen werden soll
- Schlüsselwort: **operator** und Operator (hier „=“) -> sozusagen der Name der Funktion
- **DTYP** = Datentyp des **rechten** Operanden des Operators

Zuweisungsoperator - Beispiel

```
class String
{
     ...
   public:
          ...
      String& operator=(const String& einString);
};

String& String::operator=(const String& einString)
{
   groesse = einString.groesse;
   zeichenkette = new char[groesse];
   strcpy(zeichenkette, einString.zeichenkette);
   return *this;
}
```

Zuweisungsoperator - Regeln

- Objekte müssen immer als **const Referenzparameter** übergeben werden
- Rückgabewert muss **Referenz** auf das aktuelle Objekt zurückliefern

```
String s1(40);
String s2;
s2.operator=(s1);
s2=s1;
```

Wichtige Regel

Klassen die Speicher für einige oder alle Attribute dynamisch (mit new oder new[]) anfordern, müssen einen Destruktor, Kopierkonstruktor und Zuweisungsoperator implementieren.

Überladen von Operatoren

- Operatoren so umdefinieren, dass sie auf eigene Klassenobjekte anwendbar sind
- Vereinfacht Programmcode, lesbarer
- Klassen bekommen Funktionalität von Standarddatentypen

```
Bruch a = Bruch(12, 7);
Bruch b = Bruch(13, 5);

Bruch c = a + b;        // + überladen, so dass
                        Brüche addiert werden,
                        ohne konkreten
                        Funktionsaufruf
```

Additionsoperator

- **Fast jede** vorhanden **Operatorfunktion** kann **überladen** werden!!!

```
const Typ
operator+ (const TYP& other) const;

Typ&
operator+= (const TYP& other);
```

Überladbare Operatoren

- **Arithmetische Operatoren:**
  - Addition '+', Subtraktion '-', Multiplikation '*', Division '/' und Modulo '%'
  - Inkrement '++', Dekrement '--'
- **Logische Operatoren:**
  - Und '&&', Oder '||', Nicht '!'
- **Vergleichsoperatoren:**
  - Gleichheit '==', Ungleichheit '!=', Kleiner '<', Größer '>', Kleiner gleich '<=', Größer gleich '>='
- **Wertzuweisung:**
  - '='
  - +=', -=, *=, /=, %=, ^=, &=, |=, >>=, <<=

Weitere überladbare Operatoren

- Adressoperator '&'
- Dereferenzierungsoperator bei Methoden '->*'
- Dereferenzierungsoperator '*'
- Funktionsaufruf '()'
- Indizierung '[ ]'
- Sequentielle Auswertung ','
- Speicheranforderung 'new'
- Speicherfreigabe 'delete'
- Typkonvertierung '(typ)'

Nicht überladbar

- Bedingungsoperator ':?'
- Dereferenzierungsoperator für Zeiger auf Methoden '.*'
- Gültigkeitsoperator, Klassenspezifizierung '::'
- Punktoperator '.'
- Speicherbedarf ermitteln: 'sizeof'

Operatoren überladen

- **Als globale Funktion**: `operator+(x,y)`

- **Als Methode**: `klasse.operator+(y)`
  - Folgende Operatoren **müssen** mit Methoden überladen werden:
    - Zuweisungsoperator =
    - Funktionsaufruf ()
    - Index []
    - Zeigeroperator ->

Addition - Überladen

```
class Bruch {
   ...
   const Bruch operator+(const Bruch& right) const;
};

const Bruch
Bruch::operator+(const Bruch& right) const
{
   Bruch result;
   ...
   return result;
}
int main() { Bruch a, b;
   Bruch summe = a + b;
}
```

Addition - Überladen

```
class Bruch {
   ...
   Bruch& operator+=(const Bruch& right);
};

Bruch& Bruch::operator+=(const Bruch& right)
{
   ...
   return *this;
}

int main() {
  Bruch a, b;
  a += b;
}
```

Hinweise

- **Operator-Funktionen:**
  - Keine neuen Operatoren erstellbar
  - Wirkung auf Standard Datentypen bleibt erhalten
  - Überladene Funktionen sollten sich an der ursprünglichen Funktion des Operators orientieren (z.B. die Funktion operator+ sollte keine Subtraktion darstellen)
- **Arbeiten mit Operatoren:**
  - Rangfolge bleibt (z.B. Punkt vor Strich)
  - Anzahl der Operanden eines Operators ist unveränderbar