

Überladung & Operatoren

```
elif operation == "MIRROR_Z":
    mirror_xd.use_x = False
    mirror_xd.use_y = False
    mirror_xd.use_z = True
```

```

    Selection at the end - add back the deselected mirror modifier object
    mirror_ob.select= 1
    modifier_ob.select=1
    bpy.context.scene.objects.active = modifier_ob
    print("Selected" + str(modifier_ob)) # modifier ob is the active ob
    mirror_ob.select = 0

```

```

$ cat get_ip.sh
#!/bin/bash
get_ip() {
    curl -s $1 | grep -oE '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' | head -n 1
}
get_ip http://www.ubuntu.com

```

The terminal output shows the IP address 192.168.1.1.

Überladen von Funktionen

Mehrere Konstruktoren mit dem selben Funktionsnamen

- nur unterschiedliche Parameter (Typ bzw. Anzahl)

Grundsätzliche Eigenschaft von C++

- Alle Funktionen dürfen **überladen** werden („*overloading*“)

Compiler entscheidet auf Grund Parameteranzahl und Typ welche Funktion aufgerufen wird

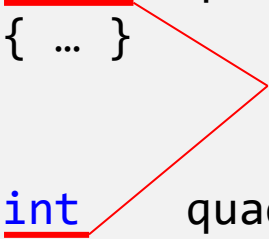
Unterscheidung nur durch Rückgabebetyp ist **nicht** zulässig!

```
int    quadrat(int i)
{ ... }

double quadrat(double d)
{ ... }

Bruch  quadrat(Bruch b)
{ ... }

int    quadrat(Bruch b)
{ ... }
```



Fehler!

Default Argumente

Angegebene Default Werte werden vom Compiler automatisch eingesetzt falls zu wenig Parameter übergeben werden

Dadurch müssen z.B. bei Bruch nur 2 Konstruktoren als Funktion implementiert werden

Default wird nur in der Deklaration definiert

- Implementierung erwartet einen Parameter

Für Klassenmethoden aber auch normale Funktionen

Auch mehrere Argumente als Default Argument möglich

Vorteile:

- Weniger Programmierarbeit
- Weniger Programmcode
- Erhöht die Konsistenz (weniger Fehler)

Potentieller Nachteil

- Performance

```
class Bruch
{
    ...
public:
    Bruch(int = 0);
    Bruch(int, int);
    ...
};
```

Default ist 0

```
Bruch::Bruch(int i)
{
    /* initialisiere mit i 1-tel */
    zaehler = i;
    nenner = 1;
}
```

Operatoren für Klassen (I)

Standard Operatoren können für eine neue Klasse überladen werden

- Z.B. +, -, *, /, <, >, =, etc.

Die Klasse muss aber die jeweiligen Implementierungen mitbringen

```
void action(Bruch wert)
{
    Bruch a;
    Bruch b = 1000;

    if (wert < b)
    {
        a = wert + b;
    }

    ...
}
```

Operatoren für Klassen (II)

Deklaration als Teil der öffentlichen Schnittstelle

Funktion wird als Methode des ersten Objekts aufgerufen

Beispiel: $a + b$

- Der Operator $+$ ist eine Methode des Objekts a und bekommt b als Parameter
- Entspricht also: `a.operator+(b)`

Bei einstelligen Operatoren gäbe es keinen Parameter.

Rückgabewert muss der Operation entsprechen

- $+$ erzeugt ein Objekt des gleichen Typs
- $<$ liefert einen boolschen Wert (true oder false)

```
class Bruch
{
    ...

    /* oeffentliche Schnittstelle */
public:

    /* Klassen Operatoren */
    Bruch operator + (Bruch);
    bool operator < (Bruch);

    ...
};
```

Operatoren für Klassen (III)

Datenkapselung in C++ ist typbezogen (Klasse)

- Daher Zugriff auf die privaten Attribute des Objekts b möglich

```
Bruch Bruch::operator + (Bruch b)
{
    int neuZaehler, neuNenner;

    neuZaehler = (zaehler * b.nenner) +
                (b.zaehler * nenner);
    neuNenner = nenner * b.nenner;

    return(Bruch(neuZaehler, neuNenner));
}

bool Bruch::operator < (Bruch b)
{
    return(zaehler * b.nenner <
           b.zaehler * nenner);
}
```

- Die für fundamentale Datentypen vordefinierten Operatoren werden nicht automatisch auf abstrakte Datentypen erweitert
 - Ausnahme ist der Zuweisungsoperator
- Die Menge möglicher Operatoren ist festgelegt
 - Kann nicht erweitert werden
- Priorität, Syntax und Auswertungsreihenfolge der Operatoren sind festgelegt
 - Z.B. Multiplikation vor Addition
- Operatoren sollten so implementiert sein, dass sie tun, was man erwarten würde
 - Also dem was man mit ihnen bei den fundamentalen Datentypen verbindet
 - Z.B. ein $+$ ist eine Addition, keine Subtraktion
 - Z.B. $x+y$ verändert weder x noch y
- $x=x+1$, $x+=1$ und $x++$ nutzen 3 verschiedene Operatoren
 - Das Verhalten sollte aber identisch sein – um Verwirrung zu vermeiden (Thema Lesbarkeit)

Operator: „=“ (**Zuweisungsoperator**)

Default Zuweisungsoperator

- Argument: Objekt der gleichen Klasse
- Falls nicht explizit spezifiziert, vom Compiler hinzugefügt:
- Jedes Attribut des „Argument“ Objekts wird dem entsprechenden Attribut des „Ziel“ Objekts zugewiesen.
 - `a.zaehler = wert.zaehler`
 - `a.nenner = wert.nenner`

Aufpassen bei Zeigern als Attribute

```
void action(Bruch wert)
{
    Bruch a;

    a = wert;

    ...
}
```


- Funktionen können **überladen** werden.
 - Ein Funktionsname kann mehrfach verwendet werden, solange sich Parameter-anzahl oder Parametertyp unterscheiden.
- Für die Argumente von Funktionen können **Default Argumente** definiert werden.
 - Werden verwendet, wenn Parameter beim Funktionsaufruf nicht übergeben werden.
- Für die Objekte einer Klasse können Operatoren überladen werden
 - Werden mit dem Schlüsselwort `operator` deklariert.
 - **Default Zuweisungsoperator** für jede Klasse vordefiniert - Zuweisung komponentenweise.



Referenzen & Copy Konstruktor

```

if operation == "Mirror_Y":
    mirror_mod.use_x = False
    mirror_mod.use_y = True
    mirror_mod.use_z = False
elif operation == "Mirror_Z":
    mirror_mod.use_x = False
    mirror_mod.use_y = False
    mirror_mod.use_z = True

#selection at the end -add back the deselected mirror modifier object
mirror_ob.select= 1
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier ob is the active ob
#mirror_ob.select = 0
#new = bpy.context.selected_objects[0]
#new.data.object.name = "new"

```

Typumwandlungen

Typumwandlungen

- Implizit (durch den Compiler)
- Explizit (durch den Programmierer)

Anweisung `b = 1000` entspricht auch:

- `b = (Bruch)1000` (Cast-Notation)
- `b = Bruch(1000)` (funktionale Notation)

Was passiert dabei:

- Erzeuge temporäres Objekts der Klasse `Bruch` und initialisiere mit dem Wert 1000 (Ausführung des entsprechenden Konstruktors)
- Zuweisung des temporären Objekts an Objekt `b`
- Zerstörung des temporären Objekts (und Ausführung des Destruktors).

Entsprechender Konstruktor muss existieren!

```
void action(Bruch wert)
{
    Bruch b;

    ...

    b = 1000;

    ...
}
```

Referenzen (I)

Funktionsaufrufe übergeben normalerweise Kopien der Parameter („*Call-by-Value*“ Mechanismus)

Beispiel

- `b + c` (entspricht: `b.operator+(c)`)
- Bei der Übergabe wird eine temporäre Kopie des Objekts `c` angelegt
- Nach Addition dann gleich wieder Zerstörung des temporären Objekts (Destruktor)

Resultat: Laufzeitnachteile

- auch wenn eine Kopie gar nicht notwendig wäre, da wir `c` nicht ändern.

Zeiger sind auch keine Lösung

- Verändert den Aufruf: `b + &c`

Neues Sprachmittel: **Referenz**

- Eine als Referenz (**&**) deklarierte Variable ist eine alternative Bezeichnung (Alias) für ein existierendes Objekt
- Hat den gleichen Typ wie das ursprüngliche Objekt (also kein Zeiger!)

Beispiel:

```
int a;  
int& b = a;
```

Verwendung von `b` ist gleichbedeutend mit `a`

Referenzen (II)

Referenzen können auch bei Parametern von Funktionen verwendet werden

- Verhindert das Anlegen der temporären Kopien

Zusätzlich kann man die Referenz als Konstante definieren (`const`)

- Verhindert das Verändern des Parameters

Auch der erste Operand kann als Konstante definiert werden („*Constant Member Function*“)

- Schlüsselwort `const` zwischen Parameterliste und Funktionskörper

```
Bruch Bruch::operator + (const Bruch& b)
const
{
    int neuZaehler, neuNenner;

    neuZaehler = (zaehler * b.nenner) +
                 (b.zaehler * nenner);
    neuNenner = nenner * b.nenner;

    return(Bruch(neuZaehler, neuNenner));
}
```

Schlüsselwort: `this`

Steht in jeder Methode als Variable zur Verfügung

Zeiger auf das Objekt für das die Methode aufgerufen wurde

- Bei einer Operator Funktion steht `*this` also für den ersten Operanden

Typische Anwendung: Rückgabewert beim Zuweisungsoperator

- Erlaubt Verkettung von Zuweisungen:

`x = y = 10;`

- `=` ist ein rechtsassoziativer Operator

- Verarbeitung von rechts nach links

`x = (y = 10);`

- Entspricht: `x.operator=(y.operator=(10));`

Rückgabewert ist eine Konstante & Referenz

- Referenz: vermeidet temporäre Objekte
- Konstante: verhindert: `(x = y) = 10;`

```
const Bruch&
```

```
Bruch::operator = (const Bruch& b)
```

```
{
```

```
    /* Zuweisung an sich selbst? */
```

```
    if (this == &b)
```

```
{
```

```
        return (*this);
```

```
}
```

```
    zaehler = b.zaehler;
```

```
    nenner = b.nenner;
```

```
    return (*this);
```

```
}
```

Copy Konstruktor

Argument: Objekt der eigenen Klasse

`<Klasse> (<Klasse>&)`

Falls nicht explizit spezifiziert, vom Compiler hinzugefügt:

- Jedes Attribut des „Argument“ Objekts wird dem entsprechenden Attribut des neuen Objekts zugewiesen.
 - `a.zaehler = wert.zaehler`
 - `a.nenner = wert.nenner`

Aufpassen bei Zeigern als Attribute

```
...  
  
void test(Bruch wert)  
{  
    Bruch a = wert;  
  
    ...  
}  
  
...
```


Zuweisungen vs. Copy Konstruktor

Zuweisungsoperator und Copy Konstruktor sind zwei verschiedene Aktionen

Default Implementierung ist identisch, aber kommen zu unterschiedlichen Zeitpunkten zum Einsatz.

- Copy Konstruktor bei der Initialisierung
- Zuweisungsoperator bei einer späteren Zuweisung

Beide können überladen/spezifiziert werden

- Sollten sich aber weiterhin gleich verhalten

```
void action(Bruch wert)
{
    Bruch a;
    Bruch b = wert; ← Copy Konstruktor

    a = wert; ← Zuweisungsoperator

    ...
}
```


- Bei einer **Typumwandlung** wird automatisch der passende Konstruktor aufgerufen.
- Durch die Angabe von **&** in der Deklaration wird eine **Referenz** deklariert
 - Eine Referenz ist eine „zweiter Name“ für ein existierendes Objekt.
 - Verwendung von Referenzen bei der Deklaration von Parametern und Rückgabewerten verhindert das Anlegen von temporären Kopien.
- Ein Konstruktor, der ein neues Objekt anhand eines existierenden Objekts gleichen Typs initialisiert, nennt man **Copy Konstruktor**.
 - Es existiert ein Default Copy Konstruktor, der komponentenweise kopiert.



Dynamischer Speicher



```
def _operation == "Mirror_Y":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = True  
    mirror_mod.use_z = False  
elif _operation == "Mirror_Z":  
    mirror_mod.use_x = False  
    mirror_mod.use_y = False  
    mirror_mod.use_z = True  
  
#selection at the end -add back the deselected mirror modifier object  
mirror_ob.select= 1  
modifier_ob.select=1  
bpy.context.scene.objects.active = modifier_ob  
print("Selected" + str(modifier_ob)) # modifier ob is the active ob  
#mirror_ob.select = 0  
#new = bpy.context.selected_objects[0]  
#new.data.object.name = "mirror_ob"  
#new.parent = mirror_ob
```

Dynamische Speicherverwaltung (I)

C	
Funktion	Verwendung
<code>malloc()</code>	Allokiere dynamisch Speicher
<code>free()</code>	Gebe dynamisch allokierten Speicher frei

Kennen keine Objekte, nur rohen Speicher.

C++	
Operator	Verwendung
<code>new</code>	Erzeuge dynamisch ein Objekt
<code>delete</code>	Zerstöre ein dynamisch erzeugtes Objekt

Funktioniert für Klassen aber auch für Standard Datentypen (z.B. `int`, `char`).
Nutzen Konstruktoren und Destruktor.
Operatoren (!), d.h. können überladen werden.

Dynamische Speicherverwaltung (II)

Operator `new`

- Verwendung: `new <type> ...`
- Liefert Zeiger auf erzeugtes Objekt / Element
- Initialisierung bei Erzeugung möglich
 - Soweit von Konstruktoren unterstützt
- Erzeugung von Arrays
 - Ein- oder mehrdimensional
 - Konstruktor wird für jedes Element aufgerufen.
- Zeiger `== NULL` zeigt Fehler (wie bei `malloc()`)

Operator `delete`

- Verwendung: `delete <Ptr>`
- Arrays müssen mit `delete [] <arrPtr>` freigegeben werden!

```
Bruch *a, *b, *c;

a = new Bruch;
b = new Bruch(1, 3);
c = new Bruch[10];
if ((a==NULL) || (b==NULL) || (c==NULL))
{
    fprintf(stderr, "kein Speicher\n");
    exit(1);
}

...

delete a;
delete b;
delete [] c;
```

Zeiger als Attribute einer Klasse (I)

```
class Beispiel
{
private:
    struct inhalt* ptr;
public:
    Beispiel();
    ~Beispiel();
};
```

```
Beispiel::Beispiel()
{
    ptr = new struct inhalt;
    ...
}

Beispiel::~~Beispiel()
{ delete ptr; }
```

Grundsätzlich möglich

- Allokation des dynamischen Speichers im Konstruktor (oder den Konstruktoren)
- Freigabe erfolgt im Destruktor

Aber **Achtung:**

Default Copy Konstruktor und Default Zuweisungsoperator sind hier nicht mehr ausreichend

- Resultieren in inkonsistenten Objektzuständen und nicht mehr freigegebenem Speicher

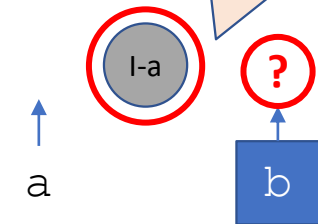
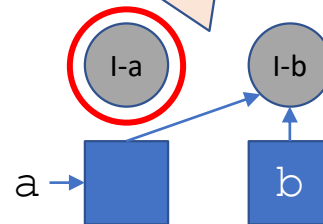
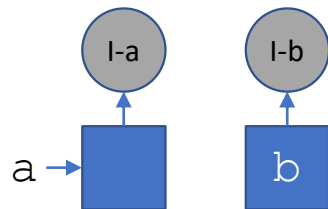
Zeiger als Attribute einer Klasse (II)

```
Beispiel *a = new Beispiel;
```

```
Beispiel b;
```

```
*a = b;
```

```
delete a;
```



Default Zuweisungsoperator kopiert nur die Attribute, d.h. der Zeiger wird kopiert. *a* und *b* verwenden ab sofort beide „I-b“. Der Zugriff auf „I-a“ geht verloren. Der zugehörige Speicher kann auch **nicht mehr freigegeben** werden.

Der Destruktor des Objekt von *a* gibt den Speicher frei auf den sein Zeiger zeigt (d.h. „I-b“!). Der Zeiger des Objekts *b* zeigt auf einen **ungültigen** Speicherbereich!

Bei Verwendung von Zeigern mit dynamischem Speicher als Attribut muss der Copy Konstruktor und Zuweisungsoperator überschrieben werden!
(Erstellung von Kopien des Inhalts, evtl. Speicherfreigabe und Neuallokation (bei Änderung des Speicherbedarfs))

- Zur **dynamischen Speicherverwaltung** werden in C++ die Operatoren `new` und `delete` eingeführt.
 - Für Arrays müssen `new` und `delete` jeweils mit eckigen Klammern aufgerufen werden.
 - `new` und `delete` lösen den Aufruf der Konstruktoren und des Destruktors aus.

