Friends

- C++ ermöglicht, einzelne Funktionen oder ganze Klassen zu friends zu erklären.
- Friendfunktionen haben Zugang zu den privaten Daten einer Klasse.
- Sie haben kein this.
- Bekommen sie aber ein Objekt der Klasse, deren Freund sie ist, als Parameter, hat sie Zugang zu dessen privaten Daten.
- Eine Klasse kann friend-vereinbarungen enthalten, sie ermöglicht dann den Freunden den Zugang zu ihren privaten Daten.
- Beispiel dazu kommt später.

Friends

- Einzelne Funktionen können als friend deklariert werden.
- Klassen können als friend deklariert werden, dann haben alle Funktionen der befreundeten Klasse Zugang zu den privaten Daten der Klasse, die die Frienddeklaration enthält.

Operatorüberladung

- C++ gestattet es, die Funktionalität von Operatoren der Sprache für Klassen zu redefinieren.
- Dabei bleiben die grundlegenden Eigenschaften der Assoziativität, der Anzahl der Operanden und der Operatorpriorität erhalten.
- Man spricht von Operatorüberladung mit Hilfe von Operatorfunktionen.
- Die überladene Funktionalität und die originale Funktionalität sollten intuitiv zusammenpassen.

Operatorüberladung

Überladbare Operatoren

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	-	<<	>>	==	!=	&	||
+=	-=	/=	%=	^=	&=	|=	*=
<<=	>>=	[]	()	->	->*	new	delete

Operatorüberladung

- Nicht überladbare Operatoren sind:

  `::  .*  .  ?: sizeof`

- Es können keine neuen Operatoren kreiert werden.
- Es gibt zweierlei Möglichkeiten Operatorfunktionen zu schreiben:
  - Memberfunktion
  - Externe Funktion (oft als friend)

Operatorüberladung mit Memberfunktion

- Operatorfunktionen als Member haben folgenden Aufbau:
  - Unärer Operator

    ```
    <ret_type> Klasse::operator @ ()
    ```

  - Binärer Operator

    ```
    <ret_type> Klasse::operator @ (<par_type> op2)
    ```

Operatorüberladung mit Memberfunktion

```
class Fraction
{
public:
 . . .
 Fraction operator+(const Fraction& Second) const;
 . . .
};
```

Operatorüberladung mit Memberfunktion

```
Fraction Fraction :: operator+ (const Fraction &
Second)const
{
  long Factor = gcf (Denom,Second.Denom);
  long Mult1  = Denom / Factor;
  long Mult2  = Second.Denom / Factor;
  long NumRes = Numerator * Mult2
              + Second.Numerator * Mult1;
  long DenRes = Denom * Mult2;
  return Fraction(NumRes, DenRes);
}
```

Es wird ein neues Objekt erzeugt und Auf den Stack gelegt

Operatorüberladung mit Memberfunktion

```
// Aufruf der Operatorfunktion
Fraction fa(3,4);
Fraction fb(3,5);
Fraction fc;
// impliziter Aufruf
fc=fa+fb;
// oder Aufruf wie klassische Funktion
fc=fa.operator+(fb);
```

Wichtig: **Der erste Operand ist von der eigenen Klasse.**

Operatorüberladung mit Memberfunktion

- Die Funktion operator+ schließt die Funktion operator+= nicht ein.

```
Fraction& Fraction::operator+= (const Fraction & Second)
{
  long Factor = gcf (Denom,Second.Denom);
  long Mult1  = Denom / Factor;
  long Mult2  = Second.Denom / Factor;
  Numerator = Numerator * Mult2
            + Second.Numerator * Mult1;
  Denom = Denom * Mult2;
  return *this;
}
```

```
// Aufruf:
  fa+=fb;
```

Es wird die Referenz
Auf das bereits existierende
Objekt zurückgegeben.

Operatorüberladung mit Memberfunktion

- Man beachte den Unterschied bei den Returnwerten:
  - `Fraction  operator+ (const Fraction & Second)const;`
  - `Fraction& operator+= (const Fraction & Second);`
- Der Operator + erzeugt ein neues Objekt und legt dieses als Returnwert auf den Stack.
- Der Operator += modifiziert das existierende Objekt des 1. Operanden und gibt eine Referenz darauf zurück.

Operatorüberladung mit Memberfunktion

- Unäre Operatoren werden analog gebaut, haben aber keinen Parameter, weil es keinen zweiten Operanden bei der Operation gibt.

```
// Natuerlich muss die hier implementierte Funktion
// In der Klasse deklariert sein.

Fraction Fraction::operator-() // negatives VZ
{
    long n=-Numerator;
    long d=Denom;
    return Fraction(n,d);
}
```

Es wird ein neues Objekt erzeugt und Auf den Stack gelegt

```
// Aufruf:
  fa=-fa;
```

Operatorüberladung mit Memberfunktion

```
class Fraction
{
    public:
    Fraction (){Numerator = 0; Denom = 1;};
    Fraction ( long Num, long Den=1);
    friend ostream& operator<<(ostream& OS,Fraction F);
    Fraction operator+ (const Fraction & Second) const;
    Fraction&operator+=(const Fraction & Second);
    Fraction operator-();
    private:
    static long gcf(long First,long Second);
    long Numerator,
   Denom;
};
```

Operatorüberladung mit externer (friend) Funktion

- Operatorfunktionen als externe Funktionen haben folgenden Aufbau:
  - Unärer Operator

    ```
    <ret_type> operator @ (<class_type> param)
    ```

  - Binärer Operator

    ```
    <ret_type> Klasse::operator @ (<par_type1> op1,
                                   <par_type2> op2)
    ```

- class_type bzw. par_type1 oder par_type2 müssen von einem Klassendatentyp sein.

Operatorüberladung mit externer (friend) Funktion

- Somit hat man die Wahl, ob man eine Operatorfunktion als Member oder als externe Funktion implementiert.
- Ist der erste Operand nicht von der eigenen Klasse, so muss der Operator durch eine externe Funktion überladen werden.
- Operatorfunktion zu () oder [] können nur als Memberfunktionen geschrieben werden.
- Statische Funktionen können keine Operatoren überladen.

Operatorüberladung mit externer (friend) Funktion

```
class Fraction
{
public:
 Fraction (){Numerator = 0; Denom = 1;};
 Fraction ( long Num, long Den=1);
 friend ostream& operator<<(ostream& OS,Fraction F);
 friendFriend Fraction operator+ (const Fraction & first,
                      const Fraction & second) const;
 friend Fraction&operator+=(Fraction & first,
                          const Fraction & Second);
 friend Fraction operator-(const Fraction & first);
private:
 static long gcf(long First,long Second);
 long Numerator,
 Denom;
};
```

Operatorüberladung mit externer (friend) Funktion

```
// Unärer Operator mit ext. Funktion ueberladen
Fraction operator-(const Fraction & first)
{
    return Fraction(first.Numerator*-1,first.Denom);
}

// Ueberladener Ausgabeoperator !!wow!!
// Operator bei dem linker Operand von fremdem Typ ist
ostream& operator<< (ostream& OS, Fraction F)
{
  OS << F.Numerator << '/' << F.Denom;
  return OS;
}
```

Operatorüberladung mit externer (friend) Funktion

```
int main()
{
   Fraction FA, FB(23,3), FC(2,3);
   FA = FB + FC;
   Cout << “FA: “<< FA << endl;
   FA = Fraction(8)+ FB;// + 10;
   cout << “FA: “<< FA << endl;
   FA+= Fraction(2);
   cout << “FA: “<< FA << endl;
   FA=-FA;
   cout << “FA: “<< FA << endl;
   return 0;
}
```

Konvertierungen in eigene Klasse

- Jeder Constructor, der nur einen Parameter übernimmt, ist formal ein **Kovertierungsconstructor**.
- Ein Konvertierungsconstructor konvertiert ein Objekt/eine Variable/Konstante des Parametertyps in ein Objekt der eigenen Klasse.
- Implizite aufrufe können zu unliebsamen Mehrdeutigkeiten beim Compilieren führen.
- Das Attribut **explicit** vor einem Konvertierungsconstructor verhindert implizite Aufrufe/Konvertierungen.

Konvertierungen in eigene Klasse

```
class Fraction
{
public:
   . . .
 Fraction (long Num, long Den=1); // KonvertierungsCon.
   . . .
};

// KonvertierungsContructor
Fraction :: Fraction(long Num, long Den)
{
   if (Den < 0)
   {       Num = -Num;       Den = -Den;   }
   Numerator   = Num;
   Denom       = Den ? Den : 1;
   int Factor  = gcf(Num,Den);
   if (Factor > 1)
   {  Numerator /= Factor ;   Denom     /= Factor ;  }
}
```

Konvertierungen in eigene Klasse

- Möglich werden Anweisungen der Art:

```
FA = FB+3;
```

- Das schließt aber nicht `FA= 3+FB;` ein.

Konvertierungen in fremde Klasse

- Konvertierungen von Objekten in eine andere klassen / anderen Datentyp werden mit **Konvertierungsoperatorfunktionen** relaisiert.
- Sie haben keinen Parameter.

```
Fraction::operator double(void)
{
  return double(zaehler)/nenner;
}
```

Konvertierungen in fremde Klasse

- Viele Konvertierungsmöglichkeiten bewirken viele Doppeldeutigkeiten.
- Beschränken auf die wirklich nötigen Konvertierungen

```
main()
{
   Fraction FA, FB(5,3), FC(5,3);
   double d=FA;
   cout << "FA: "<<FA<< " d: "<<d<<endl;

   return 0;
}
```

```
$ ./a.out
FA: 10/3 d: 3.33333
```

Der [ ]-operator

- Ein überladener Klammeroperator ermöglicht Indexprüfung bei Arrayzugriff vorzunehmen.
- Der Klammeroperator muss als Memberfunktion implementiert werden.

Der [ ]-operator

```
const int SizeArr = 24;

class IntArr
{
   public:
      IntArr (int Gr = SizeArr);   // Constructor
      IntArr (const IntArr&);      // Copycontsructor
      ~IntArr() {delete IA;}
      IntArr & operator = (const IntArr&);
      int&  operator [] (int i);
      const int&  operator [] (int i);
      int   getNum()const {return Size;}
      void  resize(int NewSz);     // new size
   protected:
      int   Size;                  // size of Array
      int * IA;
};
```

Der [ ]-operator

```
#include <iostream>
#include "intarr.h"
using namespace std;

static int Dummy=0;
IntArr :: IntArr (int Sz)
{
  Size = Sz;
  IA = new int [Size];
  for (int i=0; i<Sz; ++i) IA[i]=0;
}
IntArr :: IntArr (const IntArr& Other)
{
  Size = Other.Size;
  IA = new int[Size];
  for (int i=0; i<Size;++i) IA[i]=Other.IA[i];
}
```

Der [ ]-operator

```
IntArr& IntArr :: operator = (const IntArr& Other)
{
  delete IA;
  IA = new int [Other.Size];
  Size = Other.Size;
  for ( int i=0; i<Size;i++) IA[i]=Other.IA[i];
  return * this;
}
void IntArr :: resize (int NewSz)
{
  int * OldIA = IA;
  int   OldSz = Size;
  Size = NewSz;
  IA = new int[Size];
  int i;
  for (i=0; i<OldSz; ++i) IA[i]=OldIA[i];
  for (   ; i<Size ; ++i) IA[i]=0;
  delete OldIA;
}
```

Der [ ]-operator

```
int& IntArr :: operator [](  int i)
{
  if (i<Size) return IA[i];
  else
  {
    cout << "Indexfehler " << i << '\n';
    return Dummy;
  }
}
const int& IntArr :: operator [] (int i) const
{
  if (i<Size) return IA[i];
  else
  {
    cout << "Indexfehler " << i << '\n';
    return Dummy;
  }

}
```

Der [ ]-operator

```
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

#include "intarr.h"
void MemError ()
{
   cout << "Not enough memory\n";
    exit (1);
}
```

```
//HAUPTPROGRAMM
int main()
{
  set_new_handler(&MemError);
  IntArr IA (10);
  for (int i = 0; i < IA.getNum() ; ++i) IA[i]=i;
  for (int i = 0; i < IA.getNum() ; ++i) cout << IA[i] <<", ";
  cout << '\n';
  IA.resize(16);
  for (int i = 0; i < IA.getNum() ; ++i) cout << IA[i] <<", ";
  cout << '\n';
  cout << IA[100] <<'\n';
  return 0;
}
```