Überladung des Zuweisungsoperators

- der **Zuweisungsoperator =** weist die Datenmember eines rechts vom **=** stehenden Ausdrucks oder Objektes den **gleichnamigen** Datenmembern eines bestehenden Objektes derselben Klasse links vom **=** zu.

```
class zk { char *s;                    // Klasse ohne individuell definierten Zuweisungsoperator
   public:  zk( char *z = 0 ) : s( z ? strcpy( new char[ strlen(z) + 1 ], z ) : 0 ){  // Konstruktor
                cout<<"Konstruktor zk,  s = " << (s ? s : "0") << endl;  }
             void show(){ cout << "show s = " << (s ? s : "0") << endl; }
            ~zk(){ delete [] s; s = 0; cout << "Destruktor zk, s = " << (s ? s : "0")  << endl; }
            zk &operator=(const zk &zkr){ this->s = zkr.s; return *this; } //von C++ generiert !
};
void main(){ zk *s1 = new zk("HTW"),                 // Aufruf Konstruktor
             zk *s3 = new zk("TU");                  // Aufruf Konstruktor
             *s3 = *s1;                              // Aufruf interner Zuweisungsoperator
             delete s1; s1 = 0;                      // Freigabe Speicher s1->s und s3->s
             cout<<"s3->s = "<<s3->show()<<endl; // Abbruch, da s3->s nicht mehr existiert
             delete s3; s3 = 0;
}
```

- **C++** generiert **intern** für **jede** Klasse einen **passenden Zuweisungsoperator**, der bei der Zuweisung eins rechts vom Zuweisungsoperator stehenden Objektes die Werte der gleichnamigen Datenmember an das links vom Zuweisungsoperator stehende Objekt zuweist.

- Datenmember, die **Zeigervariablen** sind, zeigen nach der Zuweisung seitens des **internen Zuweisungsoperators** für **beide Objekte** auf einen **identischen Speicherbereich**.

- Im Falle eines mit **new** auf dem **Heap** reservierten Speicherbereiches besteht dann die Gefahr eines **Speicherzugriffsfehlers**, falls eines der Objekte mit **delete** freigegeben wird.

- Unabhängige Änderungen gemeinsamer Variablen seitens mehrerer Methoden provozieren fehlerhafte Ergebnisse.

- Ein **selbst definierter Zuweisungsoperator** **ersetzt** den von C++ **intern generierten Zuweisungsoperator** und hat die Aufgabe, seitens mehrerer Objekte derselben Klasse **gemeinsam referierte dynamische Speicherbereiche** auf dem Heap zu **verhindern**.

- Objekte derselben Klasse sollten bzgl. derjenigen Datenmember, die **Zeiger** sind, individuelle, **disjunkte Speicherbereiche** reservieren, wobei die Werte anfänglich identisch sind.

- Folgender **Zuweisungsoperator** garantiert **disjunkte Speicherbereiche** für **this->s** und **zkd.s** :

```
class zk { char *s;                    // Klasse mit individuell definiertem Zuweisungsoperator
  public:  zk( char *z = 0 ) : s( z ? strcpy( new char[ strlen(z) + 1 ], z ) : 0 ){ }  // Konstruktor
           ~zk(){ delete [] s; s = 0; }                                                // Destruktor

   zk &operator = ( const zk  &zkd ) { // Typ bzw. Parameter: Referenz bzgl. zk-Objekt
       if( this  &&  &zkd != this ) {   // existiert aktuelles Objekt ?  Adresse zkd ungleich this ?
          delete [] s;  s= 0;           // Freigabe this -> s des links von = stehenden Objektes *this
          if( &zkd )                    // Adresse von zkd ungleich 0, d.h. existiert zkd ?
            s = zkd.s ? strcpy(new char [strlen(zkd.s) + 1], zkd.s) : 0;  // this->s anlegen, kopieren
       }
       return *this;                    // Rueckgabe des links stehenden Objektes als *this
   }
};
void main(){ zk *s1 = new zk("HTW"),                // Aufruf Konstruktor
             zk *s3 = new zk("TU");                 // Initialisierung mit 0
             *s3 = *s1;  // s3->operator=(*s1);     // Aufruf individueller Zuweisungsoperator
             delete s1; s1 = 0;                     // Freigabe Speicher s1->s und s1
             cout<<"s3->s = "<<s3->show()<<endl; // OK:   s3->s = HTW
             delete s3; s3 = 0;
}
```

- Die Zuweisung ***s3 = *s1**; ist **äquivalent** zur Verwendung von **s3->operator=(*s1)**; , d.h. der links stehende Operator ***s3** wird in der Operatormethode **zk &operator = (const zk &zkd)** als ***this** verwendet, der rechts stehende Operator ***s1** ersetzt den Parameter **zkd**.

- Der oben definierte **operator=** funktioniert auch dann, wenn bei der Zuweisung ***s3 = *s1**; entweder **s1 == 0** oder (**s3 == 0** und **s1 == 0**) gilt. Bei **s3 == 0** können **s3** und ***s3** seitens der Methode **s3->operator=(*s1)** **keine Änderung** erfahren.

- Der **erste** Operand **this** im Test **if** (**this** && **&zkd != this**) **verhindert**, daß im Falle von ***s3 = *s1** mit **s3 == 0** eine Zuweisung erfolgt. Falls **s3 != 0** , dann **verhindert** der Ausdruck **&zkd != this** , daß eine Zuweisung ***s3 = *s3** erfolgt.

- Der Test **if** (**&zkd**) verhindert, daß auf ein **nicht existierendes zkd.s** zugegriffen wird. Falls **&zkd == 0** gilt, dann liegt eine Zuweisung ***s3 = 0**; vor. In dem Fall erhält ***s3** den Wert **0** .

- Im Falle von **this == 0** (vgl. 2. Anstrich dieser Seite) besteht der **linke** Operand ***s3** aus einem **dereferenzierten** Zeigerwert **0**.

- Obwohl die rechte Seite *s1 als (dynamisches) Objekt existieren kann, d.h. **s1 != 0**, ist eine Instantiierung von ***this** innerhalb einer Methode mit C++ **nicht** möglich, auch nicht die Zuweisung einer gültigen Speicherbereichsadresse an **this** :

```
zk &operator=(const zk &zkd){
  // Die Instantiierung eines Objektes seitens einer seiner Methoden "von
  // innen heraus" ist in C++ nicht möglich und erzeugt einen Runtime-Error:
  if(!this)
     return *new zk(zkd.s ? strcpy(new char[strlen(zkd.s)+1], zkd.s) : 0);

  if(&zkd != this){
     delete [] s; s=0;
     if(&zkd)
        s=zkd.s ? strcpy(new char[strlen(zkd.s)+1], zkd.s) : 0;
  }
  return *this;
}
```

- Das Problem, einem **nicht dynamisch instantiierten** Objekt (**Zeigerwert 0**) die Werte eines anderen Objektes derselben Klasse zuzuweisen, wird **nicht** mit dem Überladen des Zuweisungsoperators, **sondern** über ein mit **new** neu angelegtes Objekt, der **Nutzung des Kopierkonstruktors** und der **Zuweisung der Adresse** erreicht:

```
void main(){ zk *s1 = new zk("HTW"),   // Aufruf Konstruktor zk
                *s3 = 0;    // Initialisierung mit 0, nicht dynamisch instantiiert
/*ungeeignet*/ *s3 = *s1;          // wegen s3->operator=(*s1);  bleibt s3==0
/*geignet*/    s3 = new zk(*s1); // Kopierkonstruktor, Zuweisung Adresse, s3 != 0
             cout<<"s3->s = "<<s3->show()<<endl;        // s3 -> s = HTW
             delete s3; s3 = 0; delete s1; s1 = 0;
}
```

- Das Beispiel **s3 = 0**; **s3->operator=(*s1)**; verdeutlicht, daß **Zeigervariablen** auf eine Klasse, die den Wert **0** haben, ebenfalls die **Methoden** der Klasse rufen dürfen. Falls diese Methoden jedoch auf Datenmember der Klasse zugreifen, dann erfolgt ein **Runtime-Error**.

- **Methoden** einer Klasse sollten deshalb immer den Test auf **this != 0** enthalten:

```
// Rueckgabe von this->s
char *get_s(){
     return this ? (s ? strcpy(new char[strlen(s)+1], s) : 0) : 0;
}

//Schreiben von this->s mit z
void set_s(char *z=0){
     if(this) {
          delete [] s; this->s =z?strcpy(new char[strlen(z)+1], z):0;
     }
}
```

- **zk &operator=(const zk &zkd)** kann nur als **Methode einer Klasse** (hier **class zk**) definiert werden, nicht jedoch als Funktion.

- Die Rückgabe einer **Referenz auf zk** verhindert, daß bei **return *this** eine **Kopie** von ***this** erzeugt wird, wobei im Falle der Existenz des **Kopierkonstuktors** dieser dann aufgerufen würde. Das betrifft äquivalent auch die Übergabe von **zkd** als **Referenz**.

- Die folgende Deklaration ist ebenfalls **korrekt**, verursacht jedoch wegen der Erzeugung je einer **Kopie** des Parameters **zkd** und des aktuellen Objektes ***this** einen größeren Aufwand:

  zk operator = (const zk zkd) { ... **return *this**; } // **2x** Aufruf des **Copy-Konstruktors**

- Die Zuweisung ***s3 = *s2 = *s1;** ist äquivalent zu **s3 -> operator=(s2 -> operator= (*s1));**

- Aufruf des **individuellen Zuweisungsoperators** für **Instanzen** (ohne Zeiger) von **zk**:

```
void main(){ zk  h1(0),                    // Aufruf Konstruktor, s = 0
                 h2("TU"),                 // Aufruf Konstruktor
                 h3("HTW");                // Aufruf Konstruktor
```