Kopierkonstruktoren

- ein **Kopierkonstruktor** **initialisiert** die Datenmember eines bei der Deklaration neu zu instantiierenden Objektes mit den **gleichnamigen** Datenmembern eines bereits existierenden Objektes derselben Klasse, d.h. ein neues Objekt wird durch "**Klonen**" initialisiert.

```
class zk { char *s;                          // Klasse ohne individuell definierten Kopierkonstruktor:
   public:  zk( char *z = 0 ) : s( z ? strcpy( new char[ strlen(z) + 1 ], z ) : 0 ){  // Konstruktor
                cout<<"Konstruktor zk,  s = " << (s ? s : "0") << endl;
            }
    //      zk( zk &pz ) : s ( pz.s ){ }   // intern von C++ bereitgestellter Kopierkonstruktor
            void show(){ cout << "show s = " << (s ? s : "0") << endl; }
           ~zk(){ cout << "Destruktor zk, s = " << (s ? s : "0")  << endl; delete [] s; s = 0; }
}
void main(){ zk *s1 = new zk("HTW"),  // Aufruf Konstruktor
                    *s2 = new zk(*s1);      // Aufruf des internen - Kopierkonstruktors (Klonen)
}
```

- **C++** generiert **intern** für **jede** Klasse einen **passenden Kopierkonstruktor**, der die Datenmember eines neu zu instantiierenden Objektes mit den Werten der gleichnamigen Datenmember eines Objektes, welches als **Referenz-Parameter** übergeben wird, **initialisiert**.

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

- Bei der **delete** - Freigabe eines mit **new** auf dem **Heap** reservierten Speicherbereiches besteht die Gefahr eines **Speicherzugriffsfehlers** seitens des andere Objektes, welches von der Freigabe keine Kenntnis hat.

- Unabhängig von Speicherzugriffsfehlern können **falsche Ergebnisse** berechnet werden, falls Methoden der gemeinsamen Klasse unabhängig voneinander Daten im gemeinsamen Speicher verändern.

```
void main(){ zk *s1 = new zk("HTW"),      // Aufruf Konstruktor
             *s2 = new zk(*s1);           // Aufruf Kopierkonstruktor
             delete s2; s2=0;             // Freigabe s2->s und s2
             cout<<s1->show()<<endl;  // Abbruch, da s1->s nicht mehr existiert
}
```

- Ein **selbst definierter Kopierkonstruktor** **ersetzt** den **internen Kopierkonstruktor** und hat die Aufgabe, seitens mehrerer Objekte derselben Klasse **gemeinsam referenzierten dynamischen Speicher** auf dem Heap **zu verhindern**.

- Folgender selbst definierter **Copy-Konstruktor** der Klasse **zk** garantiert **disjunkte Speicher-bereiche** für **this->s** und **zkd.s** :

```
public:  zk( zk  &zkd ) : s( &zkd  &&  zkd.s ? strcpy( new char[ strlen(zkd.s) + 1 ], zkd.s ) : 0 ){
                if( &zkd ) cout<<"Kopierkonstruktor zk,  s = " << (s ? s : "0") << endl;
         }
```

- Falls **zkd** beim Aufruf des Copy-Konstruktors durch eine **dereferenzierte Zeigervariable**, z.B. **zk *s1 = 0; zk *s2 = new zk(*s1);** gerufen wird, dann kann die **Existenz** von ***s1** nur durch den Test auf **&zkd != 0** mit **if(&zkd)** überprüft werden, d.h. gilt **s1 != 0** ?

- **Kopierkonstruktoren** **"überladen"** den **Konstruktor** aufgrund ihrer **Signatur**.

- **Kopierkonstruktoren** gibt es für jede Klasse nur **einmal.**

- **Kopierkonstruktoren** müssen das Objekt, von dem die Initialisierungswerte "kopiert" werden sollen, **immer als Referenz** übergeben. Eine **Wertübergabe** führt zum **unendlichen rekursiven** Aufruf des Kopierkonstruktors bei dessen Nutzung und damit zum sofortigen **Abbruch**:

```
public:  zk( zk  zkd ) : s( &zkd  &&  zkd.s ? strcpy( new char[ strlen( zkd.s ) + 1 ], zkd.s ) : 0 ) {
                        // Abbruch mit Stack-Overflow  }
```

- Ein **selbst definierter Kopierkonstruktor** muß **neben** der Speicherplatzzuweisung und dem Kopieren der Werte dynamischer Speicherbereiche **zusätzlich auch alle** diejenigen **Datenmember** mit gleichnamigen Objektmembern initialisieren, die **keine Zeiger** und **nicht static** sind.

- **Kopierkonstruktoren** werden **automatisch** überall dort gerufen, wo die **Wertkopie** einer **Variablen** oder eines **Objektes** erzeugt wird, das geschieht bei der

  - **Wertübergabe** eines Objektes als Funktions- bzw. Methodenparameter

  - **Rückgabe** eines Objektes mit **return** als Funktions- bzw. Methodenwert

  - **Initialisieren** eines Objektes bei der Deklaration mit einem **existierenden** Objekt derselben Klasse

- Im Falle von **Referenzen** als **Parameter** oder **Funktionstypen** wird beim Aufruf **keine Wertkopie** erzeugt und damit auch **kein Kopierkonstuktor** gerufen.

- Im Falle von **Zeigervariablen** auf Objekte als Parameter oder **Zeigern** auf Objekte als Funktionstypen wird beim Aufruf nur eine **Kopie der Adresse** erzeugt, **nicht** jedoch der Kopierkonstruktor.