

Die verschiedenen Programmierparadigmen von C++

Kopierkonstruktor und Zuweisungsoperator

Software-Technik: Vom Programmierer zur erfolgreichen ...

8 Abstrakte Datentypen: Einheit von Daten und Funktionalität

8.1 Die Bedeutung von Schnittstellen

8.2 Klassen als abstrakte Datentypen

...


8.2.3 Standardfunktionalität in der Klassen-Schnittstelle

...

8.3 Generische Programmierung, 2. Teil

8.4 Ausnahmebehandlung, 2. Teil

8.5 Zusammenfassung

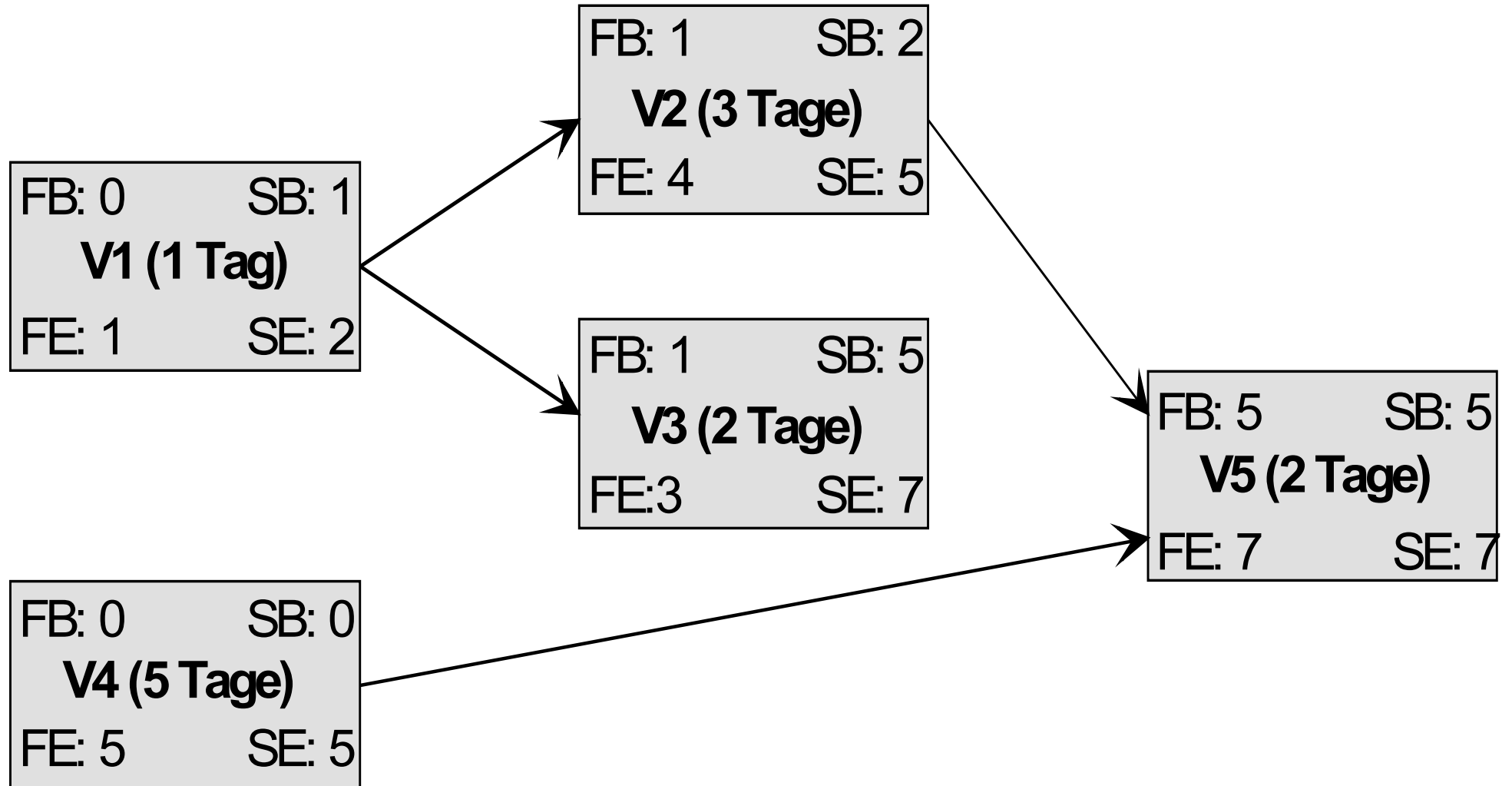
Folien mit gelben Punkten  am oberen rechten Rand sind weniger wichtiger für das Verständnis der nachfolgenden Kapitel.

Lehrbuch: 4.1.3; 4.1.6; 4.1.8

Kompendium, 3. Auflage: 9.1.2 - 9.1.4

Kompendium, 4. Auflage: 8.3; 8.4; 8.6

Beschreibung der Aufgabe / Ergebnis



Motivation von Kopierkonstruktor und Zuweisungsoperator

```
class Vorgang {  
    Vorgang** nachfolger; // Array der Nachfolger  
    int anzNachfolger;  
    Vorgang** vogaenger; // Array der Vorgänger  
    int anzVogaenger;  
    double dauer;  
public:  
    Vorgang() {  
        vogaenger = new Vorgang*[MAX]; anzVogaenger = 0;  
        nachfolger = new Vorgang*[MAX]; anzNachfolger = 0;  
    }  
};
```

Lösung, wenn Vorgänger etc.
direkt im Vorgang gespeichert
werden.

Die folgenden Folien beschreiben das Problem von Zuweisungsoperator und Kopierkonstruktor stärker aus C++-Sicht, unabhängig vom Beispiel im Buch (Abschn. 8.2.3).

Motivation von Kopierkonstruktor und Zuweisungsoperator (2)

```
typedef class Vorgang* TP_Vorgang;
```

```
class Vorgang {  
    TP_Vorgang* nachfolger; // Array von Zeiger auf Nachfolger  
    int anzNachfolger;
```

```
    TP_Vorgang* vorgaenger; // Array von Zeiger auf Vorgänger  
    int anzVorgaenger;  
    double dauer;
```

```
public:  
    Vorgang() {  
        vorgaenger = new TP_Vorgang[MAX]; anzVorgaenger = 0;  
        nachfolger = new TP_Vorgang[MAX]; anzNachfolger = 0;  
    }  
};
```

Wenn man dynamische Arraygrößen haben wollte, müsste man diese nur an den Konstruktor von Vorgang übergeben.

Motivation von Kopierkonstruktor und Zuweisungsoperator (3)

```
typedef class Vorgang* TP_Vorgang;
```

Als eigene Klasse

```
class Vorgang {
```

```
    TP_Vorgang* nachfolger; // Array von Zeiger auf Nachfolger  
    int anzNachfolger;
```

```
    TP_Vorgang* vorgaenger; // Array von Zeiger auf Vorgänger  
    int anzVorgaenger;
```

```
    double dauer;
```

```
public:
```

```
    Vorgang() {
```

```
        vorgaenger = new TP_Vorgang[MAX]; anzVorgaenger = 0;
```

```
        nachfolger = new TP_Vorgang[MAX]; anzNachfolger = 0;
```

```
    }
```

```
};
```

Motivation von Kopierkonstruktor und Zuweisungsoperator (4)

```
class Vorgang {  
    DynVorgangsArray nachfolger; // Nachfolger  
    DynVorgangsArray vorgaenger; // Vorgänger  
    double dauer;  
    /* ... */  
};
```

Statt TP_Vorgang
kann jeder beliebige
Typ verwendet werden.
Man erhält eine
Schablone für
dynamische Arrays.

```
typedef class Vorgang* TP_Vorgang;  
class DynVorgangsArray {  
public:  
    DynVorgangsArray();  
    ~DynVorgangsArray();  
    void fuegeHinzu(Vorgang* v);  
    bool istElem(const Vorgang* v) const;  
private:  
    TP_Vorgang* array;  
    int anz;  
};
```

Implementierung von DynVorgangsArray

```
DynVorgangsArray::DynVorgangsArray() {
    anz = 0;
    array = nullptr;
}
DynVorgangsArray::~DynVorgangsArray() {
    delete[] array;  array = nullptr;  anz = 0;
}

/** Kommt Vorgang *v in array vor?*/
bool DynVorgangsArray::istElem
(const Vorgang* v) const {
    for (int i=0; i < anz; ++i) {
        if (array[i] == v) {
            return true;
        }
    }
    return false;
}
```

```
/** Vorgang v wird dem dynamischen
Vorgangsarray hinzugefügt. Hierbei
muss der Speicherplatz auf dem Heap, den
arr belegt um 1 vergrößert werden bzw.
sogar erst angefordert werden. */
void DynVorgangsArray::fuegeHinzu(Vorgang*
v) {
    // Speicherplatz fuer einen mehr
    Vorgang** tmp = new TP_Vorgang[anz+1];
    // bisherigen Inhalt retten
    for (int i=0; i < anz; ++i) {
        tmp[i] = array[i];
    }
    // Neuen Eintrag eintragen
    tmp[anz] = v;    ++anz;
    // bisherigen Speicher freigeben
    delete [] array;
    // Zeiger umbiegen
    array = tmp;
}
```

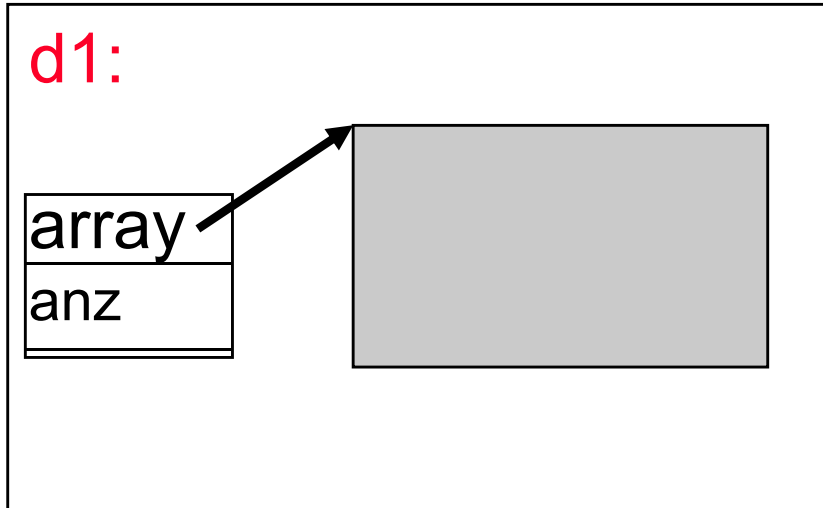

Test der Implementierung von DynVorgangsArray

```
bool testDynArray() {  
    DynVorgangsArray d1;  
    Vorgang v1;  
    Vorgang v2;  
    d1.fuegeHinzu(&v1);  
  
    bool retValue = d1.istElem(&v1) &&  
        d1.istElem(&v2)==false;  
  
    return retValue;  
}
```

```
void machNix(DynVorgangsArray d) {  
    }  
  
bool testDynArrayCopy() {  
    DynVorgangsArray d1;  
    Vorgang v1;  
    Vorgang v2;  
    d1.fuegeHinzu(&v1);  
    bool retValue= d1.istElem(&v1) &&  
        d1.istElem(&v2)==false;  
  
    machNix(d1);  
  
    return retValue;  
}
```

Programmabsturz

Speicherbelegung durch die Instanzen von DynVorgangsArray

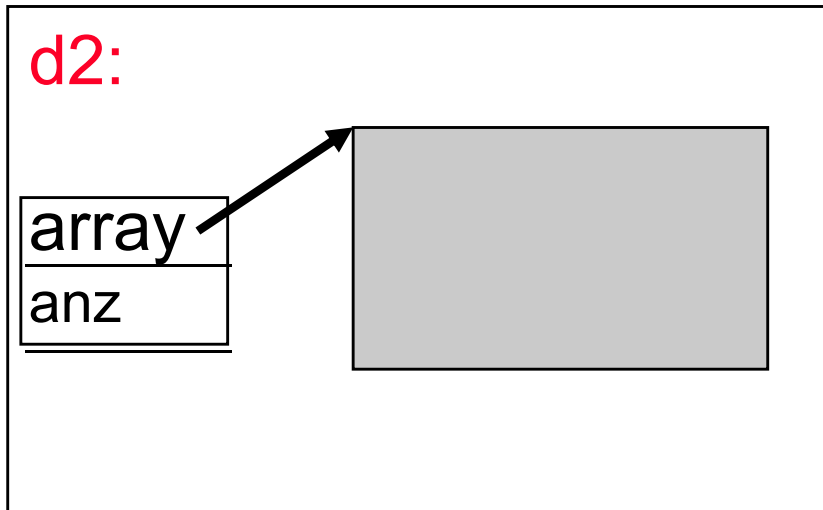


Entsprechend:

```
DynVorgangsArray d1;
```

```
DynVorgangsArray d2;
```

Das Array `array` und das Attribut `anz` gibt es also für jede Instanz von `DynVorgangsArray`, d.h. hier zweimal.



Entsprechend:

```
int i;
```

```
int k;
```

Die Bedeutung des Kopierkonstruktors

Der Kopierkonstruktor hat eine ganz besondere Bedeutung -- er wird in vielen Fällen implizit vom System verwendet, z.B. in den beiden folgenden Funktionen **f1** und **f2**:

```
void f1(Netz n) { . . . }  
Netz f2()      { Netz temp; . . .; return temp; }
```

Die Funktion **f1** hat Werteparameter **Netz n**, d.h. der Wert der Variablen (des Objektes) **n** wird in die Funktion hineinkopiert, und dazu verwendet das System den Kopierkonstruktor! Entsprechend verwendet das System in der Funktion **f2** den Kopierkonstruktor, um den Wert von **Netz temp** zurückzugeben! Zusammenfassend ist Folgendes hervorzuheben:

- Ein korrekt arbeitender Kopierkonstruktor ist für fast jede Klasse erforderlich, weil er vielfach implizit vom System verwendet wird.
- Die allgemeine Form der Deklaration eines Kopierkonstruktors für die Klasse **X** lautet: **X(const X& x);**

Standard- Copy-Konstruktor

Standardmäßig erzeugt der Compiler automatisch für jede Klasse einen Copy-Konstruktor mit dem folgenden Sourcecode.

```
Vorgang::Vorgang(const Vorgang & v2) {  
    *this = v2;  
}
```

was gleichbedeutend ist mit:

```
Vorgang::Vorgang(const Vorgang& v) {  
    dauer = v.dauer;  
    fruehanf = v.fruehanf;  
    spaetend = v.spaetend;  
    id = v.id;  
}
```

```
class Vorgang {  
    int dauer;  
    int fruehanf;  
    int spaetend;  
    int id;  
};
```

Diese Standardform ist für Vorgang ausreichend, da nicht mit Zeigersemantik (auf Heap) gearbeitet wird.

Standard- Copy-Konstruktor für DynVorgangsArray

Standardmäßig erzeugt der Compiler automatisch für jede Klasse einen Copy-Konstruktor mit dem folgenden Sourcecode.

```
DynVorgangsArray:: DynVorgangsArray(const DynVorgangsArray & d2) {  
    *this = d2;  
}
```

was gleichbedeutend ist mit:

```
DynVorgangsArray:: DynVorgangsArray(const DynVorgangsArray & d2){  
    array = d2.array;  
    anz = d2.anz;  
}
```

Diese Standardform ist **nicht** ausreichend.

Standardkopierkonstruktor für Klasse DynVorgangsArray

d1:



```
DynVorgangsArray d1;  
// in d1 5 Vorgänge eintragen  
DynVorgangsArray d2(d1); // Kopie von d1
```



d2:



```
d2.fuegeHinzu(&v1);  
Fügt in d1 ein.
```

Übung

Bauen Sie den Kopierkonstruktor und Zuweisungsoperator für diese Klasse.

```
#ifdef USE_REF_PARAM
// mit Referenzparameter geht es sogar
void machNix(DynVorgangsArray& d) { }
#else
void machNix(DynVorgangsArray d) { }
#endif
bool testDynArray() {
    DynVorgangsArray d1;
    Vorgang v1;  Vorgang v2;
    d1.fuegeHinzu(&v1);
    bool retValue = d1.istElem(&v1) &&
        d1.istElem(&v2)==false;

    machNix(d1);
    return retValue;
}
```

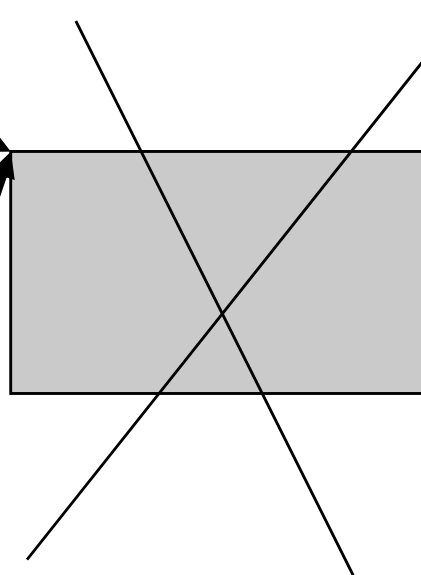
[Source-Code auf meiner Homepage](#)

Standardkopierkonstruktor: Problem 2

d1:



d2:

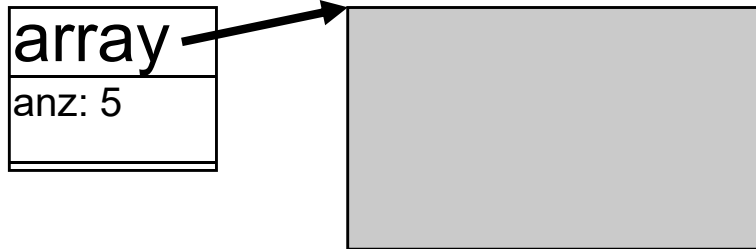


```
DynVorgangsArray d1;  
// in d1 5 Vorgänge eintragen  
if (a==b) {  
    DynVorgangsArray d2(d1);  
    ...  
} // d2 wird freigegeben  
d1.fuegeHinzu(&v7);
```

d1.array ist auch weg

Standardkopierkonstruktor: Lösung

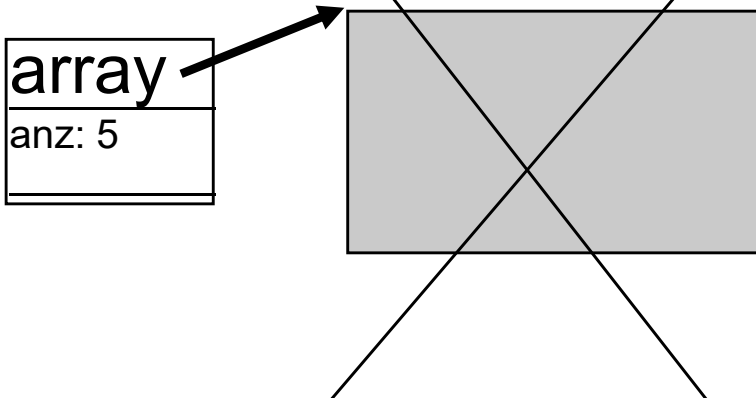
d1:



```
DynVorgangsArray d1;  
// in d1 5 Vorgänge eintragen
```

```
if (a==b) {  
    DynVorgangsArray d2(d1);  
    ...  
}
```

d2:



```
} // d2 wird freigegeben  
d1.fuegeHinzu(&v7);
```

Kopierkonstruktor für die Klasse DynVorgangsArray

```
class DynVorgangsArray {
public:
    DynVorgangsArray();
    ~DynVorgangsArray();
    DynVorgangsArray(
        const DynVorgangsArray&);
    void fuegeHinzu(Vorgang* v);
    bool istElem(const Vorgang* v)
        const;

private:
    TP_Vorgang* array;
    int anz;
};
```

```
DynVorgangsArray::DynVorgangsArray
    (const DynVorgangsArray& d2) {
    anz = d2.anz;
    array = new TP_Vorgang[anz];
    for (int i=0; i<anz; ++i){
        array[i] = d2.array[i];
    }
}
```

Implizite Erzeugung des Zuweisungsoperators

Es werden je 2 Objekte vom Typ **Vorgang** und vom Typ **DynVorgangsArray** definiert und in Zuweisungen verwendet. Für beide Typen werden die im vorangehenden verwendeten Definitionen zugrunde gelegt. Beide Zuweisungen sind syntaktisch korrekt, die Zuweisung **d2 = d1;** ist aber **inhaltlich falsch!**

```
class DynVorgangsArray { TP_Vorgang* array; . . . };
class Vorgang { double dauer; . . . };
. . .
DynVorgangsArray d1, d2;
Vorgang v1, v2;
. . .
d2 = d1; // logischer Fehler !
v2 = v1; // in Ordnung !
. . .
```

Für beide Klassen wurde kein Zuweisungsoperator explizit definiert, in dem Falle definiert -- ähnlich wie im Falle des Kopierkonstruktors -- das System implizit einen der komponentenweise kopiert, d.h. aber, es wird **flach kopiert**. Deshalb erfolgt die Zuweisung der Vorgang-Objekte korrekt, während die Zuweisung der **DynVorgangsArray**-Objekte fehlerhaft abläuft.

Definition von Zuweisungsoperatoren

```
DynVorgangsArray & DynVorgangsArray ::operator=  
                                (const DynVorgangsArray & d) {
```

```
    if (this != &d) {  
        Freigeben();  
  
        KopiereNetz(d);  
    } // if (this != &d)  
  
    return *this;  
}
```

Allgemeiner Aufbau des Zuweisungsoperators

1. Eigenzuweisung verhindern
2. Löschen des bisherigen Inhalts von *this
3. neuen Speicher anfordern
4. Kopieren des zweiten Operanden nach *this
5. Referenz auf *this zurückliefern

Definition von Zuweisungsoperatoren (2)

```
DynVorgangsArray& DynVorgangsArray::operator=  
    (const DynVorgangsArray& d2) {  
    if (this != &d2) {  
        TP_Vorgang* tmp =  
            new TP_Vorgang[d2.anz];  
        for (int i=0; i < d2.anz; ++i) {  
            tmp[i] = d2.array[i];  
        }  
        // bisherigen Speicher freigeben  
        delete [] array;  
        anz = d2.anz;  
        // Zeiger umbiegen  
        array = tmp;  
    }  
    return *this;  
}
```

Minimale Standardschnittstelle

Folgende Elemente sind für viele Klassen eine notwendige Grundausstattung:

- Standardkonstruktor: **X()**
- Kopierkonstruktor: **X(const X& x)**
- Destruktor: **~X()**
- Zuweisungsoperator: **X& operator= (const X& x)**

Die Meinungen zum Thema "Minimale Standardschnittstelle" sind allerdings uneinheitlich.

Empfehlung: Beim Entwurf einer Klasse für jedes dieser vier Elemente **prüfen, ob es definiert sein soll. Wenn ja**, entscheiden, ob das jeweils vom System erzeugte Default-Element in Ordnung ist, oder ob das Element explizit selbst definiert werden soll. **Wenn nein**, dann sollte das Element explizit unterbunden werden, siehe dazu die folgende Erläuterung.



Unterbinden von Default-Operationen

Das folgende Beispiel zeigt, wie durch Deklaration des Kopierkonstruktors und des Zuweisungsoperators im privaten Bereich die entsprechenden Funktionen unterdrückt werden können. Die zugehörigen Definitionen können entfallen!

```
... H-Datei
class X {
  private:
    ...
    X(const X& x);
    X& operator= (const X& x);
  public:
    ...
};
...
```

```
... Anwendung
X x1, x2;
...
x2 = x1; Syntaxfehler !
...
X x3(x1); Syntaxfehler !
...
```

Unterbinden von Default-Operationen

```
class DefVec {  
public:  
    DefVec() = default;  
    DefVec(int s, double d) :  
        size(s), defWert(d) {};  
    DefVec(const DefVec&) = delete;  
    ~DefVec() = default;  
    double GetDefWert(){ return defWert;}  
private:  
    int size;    double defWert;  
};
```

```
int main() {  
    DefVec v1(1, 1.6);  
    cout << v1.GetDefWert() << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

Unterbinden von Default-Operationen

```
class DefVec {  
public:  
    DefVec() = default;  
    DefVec(int s, double d) :  
        size(s), defWert(d) {};  
    DefVec(const DefVec&) = delete;  
    ~DefVec() = default;  
    double GetDefWert(){ return defWert;}  
private:  
    int size;    double defWert;  
};
```

```
int main() {  
    DefVec v1(1, 1.6);  
    cout << v1.GetDefWert() << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

Unterbinden von Default-Operationen

```
class DefVec {  
public:  
    DefVec() = default;  
    DefVec(int s, double d) :  
        size(s), defWert(d) {};  
    DefVec(const DefVec&) = delete;  
    ~DefVec() = default;  
    double GetDefWert(){ return defWert;}  
private:  
    int size;    double defWert;  
};
```

```
int main() {  
    DefVec v1(1, 1.6);  
    DefVec v3 = v1;  
    cout << v1.GetDefWert() << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

Unterbinden von Default-Operationen

```
class DefVec {  
public:  
    DefVec() = default;  
    DefVec(int s, double d) :  
        size(s), defWert(d) {};  
    DefVec(const DefVec&) = delete;  
    ~DefVec() = default;  
    double GetDefWert(){ return defWert;}  
private:  
    int size;    double defWert;  
};
```

```
int main() {  
    DefVec v1(1, 1.6);  
    DefVec v3 = v1;  
    cout << v1.GetDefWert() << endl;  
}
```

Bildschirmausgabe ?

1. Compilerfehler; Copy-Constructor verboten
2. Compiler Warnung
3. 1.6
4. 1

Bildschirmausgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

Unterbinden von Default-Operationen

```
class DefVec {  
public:  
    DefVec() = default;  
    DefVec(int s, double d) :  
        size(s), defWert(d) {};  
    DefVec(const DefVec&) = delete;  
    ~DefVec() = default;  
    double GetDefWert(){ return defWert;}  
private:  
    int size;    double defWert;  
};
```

```
int main() {  
    DefVec v1(1, 1.6);  
    DefVec v2(2, 2.6);  
    v2 = v1;  
    cout << v1.GetDefWert() << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

Unterbinden von Default-Operationen

```
class DefVec {  
public:  
    DefVec() = default;  
    DefVec(int s, double d) :  
        size(s), defWert(d) {};  
    DefVec(const DefVec&) = delete;  
    ~DefVec() = default;  
    double GetDefWert(){ return defWert;}  
private:  
    int size;    double defWert;  
};
```

```
int main() {  
    DefVec v1(1, 1.6);  
    DefVec v2(2, 2.6);  
    v2 = v1;  
    cout << v1.GetDefWert() << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.6
4. 1

Unterbinden von Operationen

```
class NewVec {  
public:  
    NewVec(int s, double d) :  
        size(s), defWert(d) {};  
    double GetDefWert(){ return defWert; }  
    void* operator new(std::size_t) = delete;  
private:  
    int size; double defWert;  
};
```

```
int main() {  
    NewVec n1(1, 1.8);  
    NewVec* p = &n1;  
    cout << p->GetDefWert() << endl;  
    NewVec* p = new NewVec(16, 16.4);  
    cout << p->GetDefWert() << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.8 16.4
4. 1.8 1.8

Unterbinden von Operationen

```
class NewVec {  
public:  
    NewVec(int s, double d) :  
        size(s), defWert(d) {};  
    double GetDefWert(){ return defWert; }  
    void* operator new(std::size_t) = delete;  
private:  
    int size; double defWert;  
};
```

```
int main() {  
    NewVec n1(1, 1.8);  
    NewVec* p = &n1;  
    cout << p->GetDefWert() << endl;  
    NewVec* p = new NewVec(16, 16.4);  
    cout << p->GetDefWert() << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler, new verboten für diese Klasse
2. Compiler Warnung
3. 1.8 16.4
4. 1.8 1.8

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 1.8 16.4
4. 1.8 1.8

Unterbinden von Operationen

```
class FVec {  
public:  
    FVec(double d) : w(d) {};  
    double Wert(int k) {return k*w;}  
    double Wert(double d) = delete;  
private:  
    double w;  
};
```

```
int main() {  
    FVec f(1.1);  
    cout << f.Wert(8) << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

Unterbinden von Operationen

```
class FVec {  
public:  
    FVec(double d) : w(d) {};  
    double Wert(int k) {return k*w;}  
    double Wert(double d) = delete;  
private:  
    double w;  
};
```

```
int main() {  
    FVec f(1.1);  
    cout << f.Wert(8) << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

Unterbinden von Operationen

```
class FVec {  
public:  
    FVec(double d) : w(d) {};  
    double Wert(int k) {return k*w;}  
    double Wert(double d) = delete;  
private:  
    double w;  
};
```

```
int main() {  
    FVec f(1.1);  
    cout << f.Wert(8.0) << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler
2. Compiler Warnung
3. 8.0
4. 8.8

Unterbinden von Operationen

```
class FVec {  
public:  
    FVec(double d) : w(d) {};  
    double Wert(int k) {return k*w;}  
    double Wert(double d) = delete;  
private:  
    double w;  
};
```

```
int main() {  
    FVec f(1.1);  
    cout << f.Wert(8.0) << endl;  
}
```

BildschirmAusgabe ?

1. Compilerfehler; darf nicht mit double aufgerufen werden
2. Compiler Warnung
3. 8.0
4. 8.8

BildschirmAusgabe ?

1. Compilerfehler
Compiler Warnung
8.0
8.8

Implizite Erzeugung von Destruktoren

Wenn in einer Klasse kein Destruktor explizit definiert ist, dann wird -- entsprechend der impliziten Definition bei den Konstruktoren -- implizit ein **Default-Destruktor** vom System definiert. So wie Default-Konstruktoren keinen Speicherplatz auf dem Heap reservieren, geben auch Default-Destruktoren keinen Speicherplatz auf dem Heap frei:

- Die implizite Erzeugung eines Default-Destruktors kann in der Regel nur dann korrekt erfolgen, wenn die Klasse keine dynamisch verwalteten Daten (d.h. in der Regel: keine Zeiger) enthält.



Werte- und Zeigersemantik beim Arbeiten mit Objekten

Wertesemantik

```
void fw (. . .) {  
    Netz n;    // Konstruktoraufruf  
    . . .  
    n.fuegeHinzu(v2);  
    . . .  
}           // Destruktoraufruf
```

Zeigersemantik:

```
void fz (. . .) {  
    Netz* pn;  
    . . .  
    pn = new Netz(0,7); // Konstruktoraufruf  
    . . .  
    pn->fuegeHinzu(v2)  
    . . .  
    delete pn;    // Destruktoraufruf  
    . . .  
}
```

Übungsaufgabe

```
private:  
    int* daten;  
    int dimension;  
};
```

```
void DoNothing(Vektor arg){ }
```

```
bool testMacheNix() {  
    Vektor v(4);  
    DoNothing(v);  
    return true;  
}
```

```
Vektor::~Vektor(){  
    delete[] daten;  
    daten = nullptr;  
}
```

7. Vorlesung; Mo. 23.11.2023 / 6. Woche

Vorlesung

[Wiederholung / Ankündigung \(20.10.2023\)](#) [tiefe und flache Kopie \(31.10.2023\)](#)
[Weitere Informationen zu Konstruktoren \(Selbststudium als Clickeraufgaben\) \(20.10.2023\)](#)

Übungsaufgaben WS 2023/24; Sprechfunk-Annotation, keine direkte Bewe

[Exercise: Classes and deep and shallow copy \(20.10.2023\)](#)

Übungsaufgaben

[Infos zu cmake \(29.09.2023\)](#)

DynVorgangsArray mit Problemen [VS 2022](#) bzw. [CMake \(20.10.2023\)](#)

Vektor mit Kopier-Problemen [VS 2022](#) bzw. [CMake \(20.10.2023\)](#)