



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG



Übung:

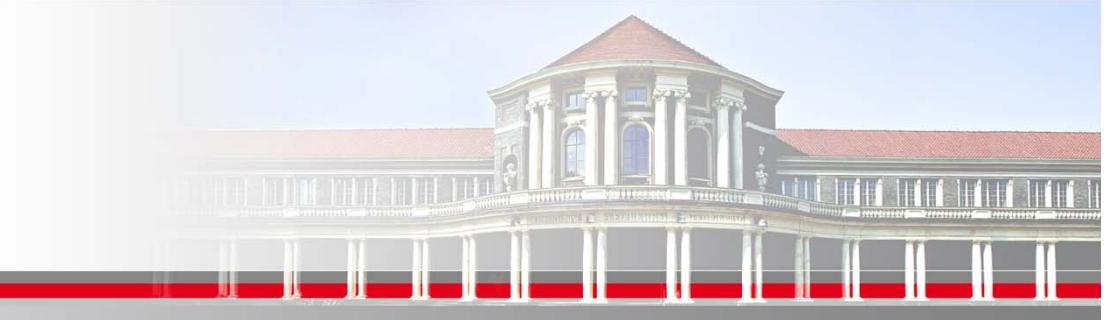
Datenvisualisierung und GPU-Computing

Programmieren in C und C++

Teil 3

Michael Vetter

michael.vetter@rrz.uni-hamburg.de



Agenda

- Klassen in C++
- Konstruktoren
- Destruktoren
- Copy-Konstruktoren
- Zuweisungs-Operator
- Vererbung
- Kapselung
- Templates
- STL



Klassen, Definition

- Allgemein:

```
class Name
{
    public:
        // Öffentliche Komponenten
        // (Konstruktoren, Methoden usw.)

    protected:
        // Geschützte Komponenten

    private:
        // Private Komponenten
};
```

Nicht vergessen!





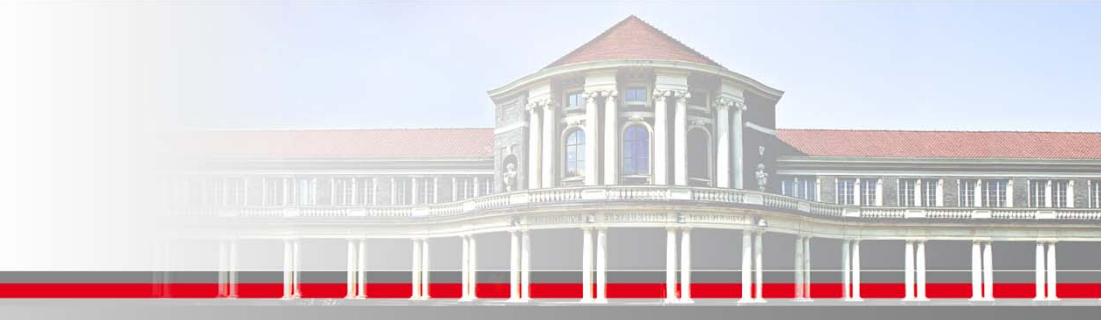
Klassen, Beispiel

```
class Vector2D
{
public:
    float x() const;
    float y() const;

    void setX(float value);
    void setY(float value);

protected:
    float mElement[2];
};
```

Zeigt dem Compiler an, dass innerhalb der Methode keine Membervariablen verändert werden dürfen.
⇒ Zusätzliche Optimierung durch den Compiler möglich.



Klassen, Programmaufbau (1)

- Klassen werden normalerweise in den *Header Dateien* deklariert
 - Klassenname.h
- Die Implementierung von Funktionen kommt in die *source Dateien*
 - Klassenname.cpp
 - Klassenname.C, Klassenname.cc, nicht verwenden

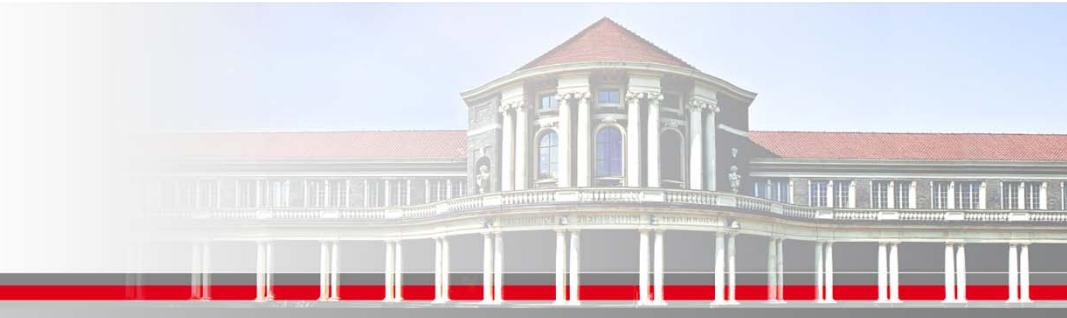


Klassen, Programmaufbau (2)

- Da sich die Implementierung nicht innerhalb der Definition befindet, muss man irgendwie die Verbindung zur jeweiligen Klasse herstellen
- Dazu dient folgende Syntax:

```
Rückgabetyp Klasse::Methode(Parameter) {  
    // Implementierung  
}
```

- Die Implementierung einer Klasse kann ohne weiteres auf mehrere Quelldateien verteilt werden



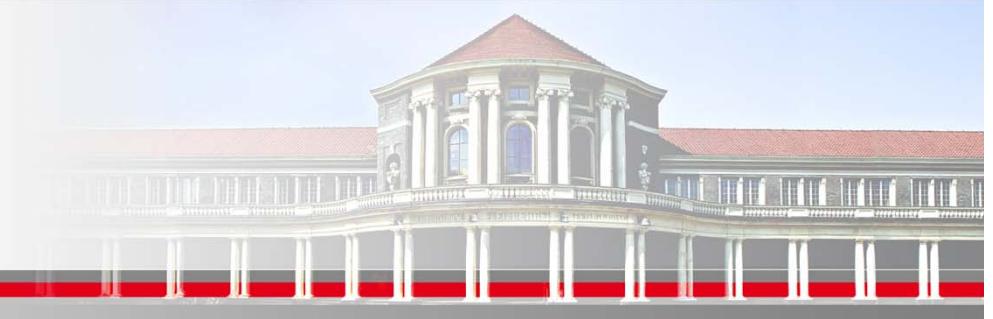
Klassen, Programmaufbau (3)

```
#include "Vector2D.h"

float Vector2D::x() const
{
    return mElement[0];
}

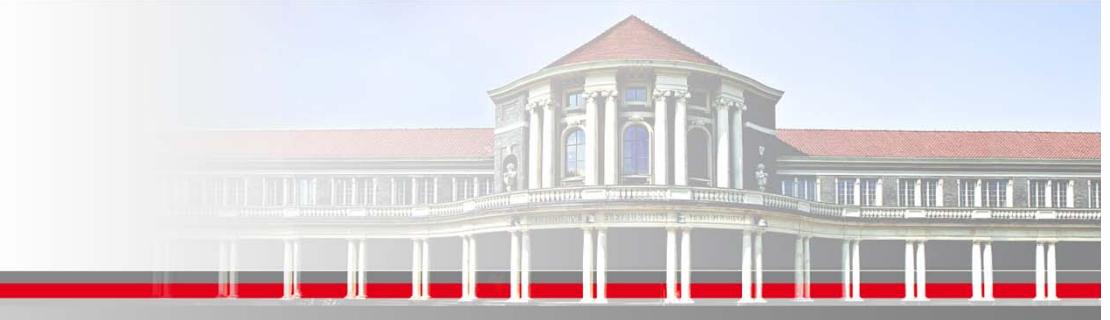
[...]

void Vector2D::setX(float value)
{
    mElement[0] = value;
}
```



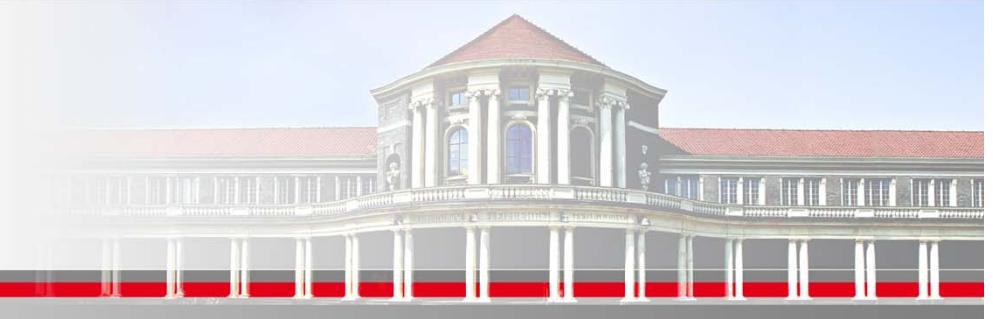
Konstruktoren (1)

- Ein *Konstruktor* ist eine spezielle Methode ohne Rückgabewert, deren Namen mit dem der Klasse übereinstimmt
- Bei der Erzeugung einer Klasseninstanz wird nach der Speicherreservierung **automatisch** der Konstruktor aufgerufen
- Definiert der Programmierer keinen eigenen Konstruktor, dann wird vom Compiler **automatisch** ein **parameterloser Default-Konstruktor** erzeugt, der aber keine Funktionalität besitzt



Konstruktoren (2)

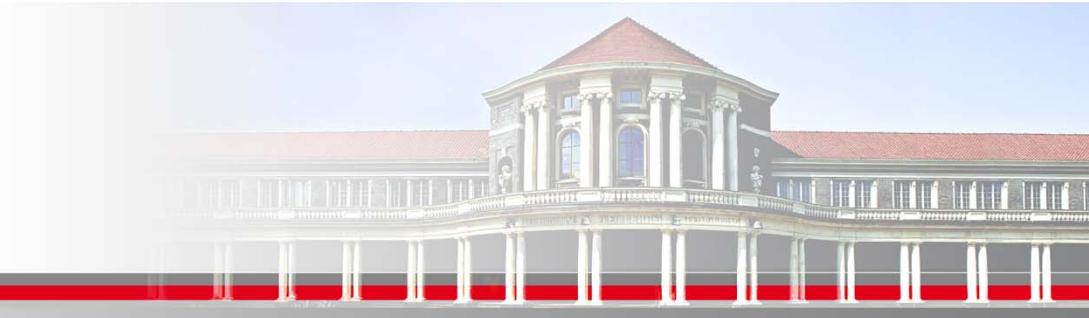
- Guideline: immer eigenen Konstruktor definieren!
- Man kann auch mehrere Konstruktoren mit unterschiedlichen Parameterlisten für eine Klasse definieren. Es ist jedoch nicht möglich, aus einem Konstruktor heraus einen anderen der gleichen Klasse aufzurufen
 - Leider ...
- Workaround: Private Initialisierungsmethode, die von allen Konstruktoren genutzt wird



Konstruktoren (3)

- Definition:

```
class Vector2D {  
public:  
    Vector2D();  
    Vector2D(float x, float y);  
  
private:  
    void init(float x, float y);  
};
```



Konstruktoren (4)

- Implementierung:

```
Vector2D::Vector2D() {
    init(0, 0);
}

Vector2D::Vector2D(float x, float y) {
    init(x, y);
}

void Vector2D::init(float x, float y) {
    mElement[0] = x;
    mElement[1] = y;
}
```



Destruktoren (1)

- Ein *Destruktor* ist das Gegenstück zu einem Konstruktor
- Wird automatisch für jedes Objekt am Ende seiner Lebensdauer aufgerufen
- Ein Destruktor ist immer parameterlos und besitzt ebenfalls keinen Rückgabewert
- Name setzt sich aus dem Klassen-Namen und einer vorangestellten Tilde zusammen
- **Destruktoren** sollten immer dann verwendet werden, wenn in einer Klasse Member-Variablen **dynamisch** angelegt werden! Andernfalls wird der reservierte Speicher nie freigegeben!



Destruktoren (2)

- Beispiel:

```
Vector2D::~Vector2D()
{
    // Bei Vector2D ist nichts freizugeben!
}

class Vector
{
protected:
    float * a;
public:
    Vector( int n ) { a = new float[n]; }
    ~Vector() { delete [] a; }
}
```

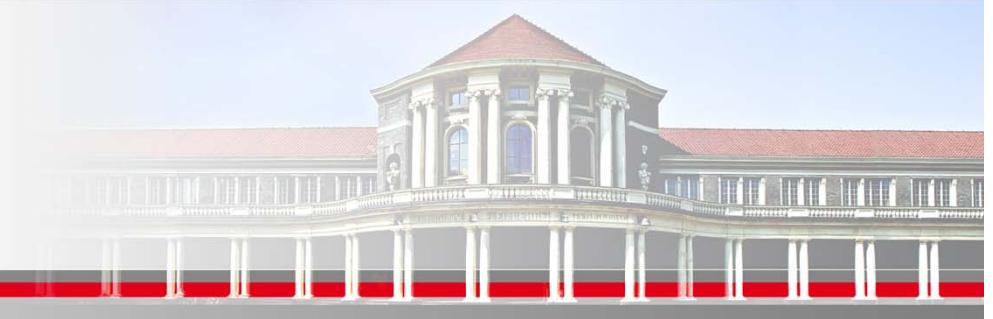


Copy-Konstruktoren (1)

- Der Copy-Konstruktor wird aufgerufen, wenn ein Objekt:
 - als Wert übergeben wird

```
CIntList f( CIntList cLL );
```
 - Bei der Initialisierung mit einem bereits existierenden Objekt

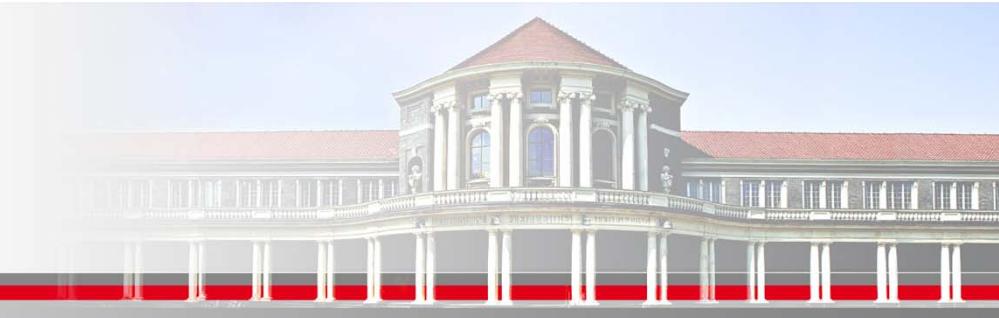
```
int main() {
    CIntList cl_11, cl_12;
    ...
    cl_12 = f( cl_11 );      // Kopie von cl_11
    CIntList cl_13 = cl_11;
}
```



Copy-Konstruktoren (2)

- von einer Funktion zurückgegeben wird

```
CIntList f( CIntList cLL ) {  
    CIntList cl_tmp1 = cLL;      // Kopie von cLL  
    CIntList cl_tmp2(cLL);      // Kopie von cLL  
    ...  
    return cl_tmp1;             // Kopie von cl_tmp1  
}
```



Copy-Konstruktoren (3)

- Deklaration

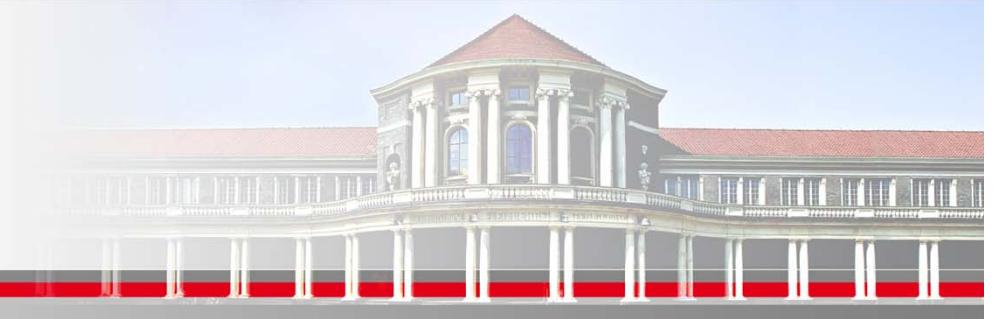
```
class CIntList {  
public:  
    CIntList();                                // „default“ ctor  
    CIntList( const CIntList &cclL ) // copy ctor  
    ...  
};
```



Copy-Konstruktoren (4)

- Definition

```
CIntList::CIntList(const CIntList &crclL):
    m_piItems( new int[crclL.m_iArraySize] ),
    m_iNumItems( crclL.m_iNumItems ),
    m_iArraySize( crclL.m_iArraySize )
{
    for ( int i_k = 0; i_k < m_iNumItems; i_k ++ ) {
        m_piItems[i_k] = crclL.m_piItems[i_k];
    }
}
```



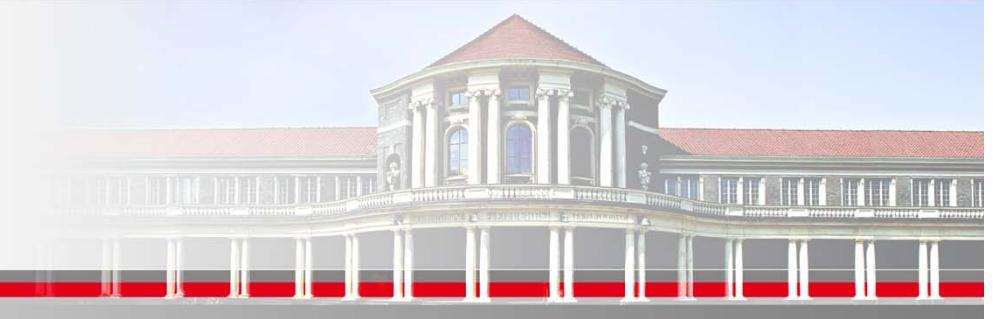
Der Zuweisungsoperator (1)

- **operator =**

- In C++ kann so ein Objekt einem anderen zugewiesen werden

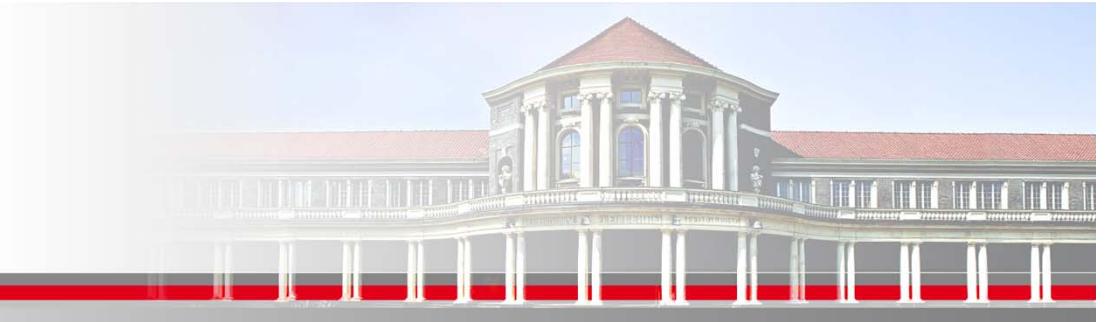
```
CIntList cl_11, cl_12;  
...  
cl_11 = cl_12;
```

- Ohne ein eindeutig Zuweisungsoperator (**operator =**), müssen die Objekte Byte-weise kopiert werden (auch *flat copy / shallow copy* genannt)



Der Zuweisungsoperator (2)

- Ohne **operator =**
 - Wenn das Objekt einen Zeiger beinhaltet, dann würde der Zeiger des neuen Objektes auf dieselbe Adresse zeigen wie das Ausgangsobjekt
 - Wird der Zeiger von **cl_11** gelöscht, dann verweist der Zeiger von **cl_12** auf eine ungültige Adresse



Der Zuweisungsoperator (3)

- Unterschied zwischen Zuweisungsoperator und Copy-Konstruktor
(am Bsp. `c1_11 = c1_12`)
 - `c1_11` ist ein bereits initialisiertes Objekt; enthält dieses einen Zeiger, so muss dieser gelöscht werden, bevor dem Objekt etwas neu zugewiesen werden kann
 - Eine Variable kann sich nicht selber zugewiesen werden → so etwas sollte man niemals machen
 - Der Code des **operator =** muss einen Rückgabewert besitzen



Der Zuweisungsoperator (4)

■ **operator =**

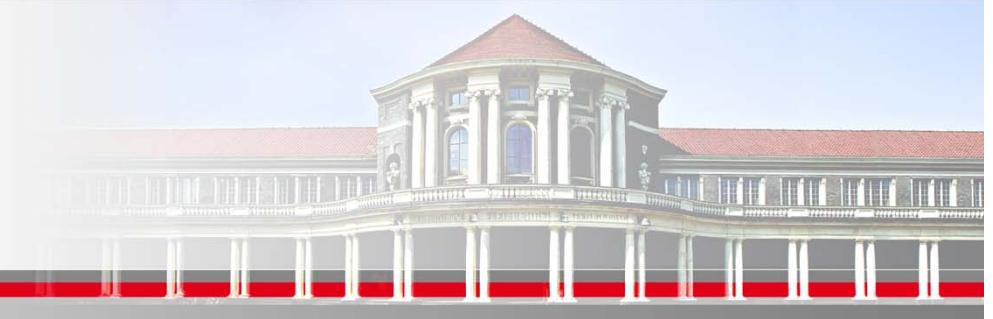
```
CIntList & CIntList::operator = ( const CIntList &crclL )
{
    // überprüfe ob Eigenzuweisung (self assignemnt)
    if ( this == &crclL )
        return *this;
    else
    {
        delete [] m_piItems;                      // Speicher freigeben
        m_piItems = new int[crclL.m_iArraySize];   // neuen Speicher holen
        m_iArraySize = crclL.m_iArraySize;         // Inst.var. kopieren

        // Kopiere crclL in das neue Array
        // zuweisen m_iNumItems
        for ( m_iNumItems=0; m_iNumItems < crclL.m_iNumItems;
              m_iNumItems ++ )
            m_piItems[m_iNumItems]= crclL.m_piItems[m_iNumItems];
    }
    return *this; // Rüchgabe CIntList
}
```



Vererbung (1)

- Eine Klasse kann Eigenschaften einer anderen Klasse durch die **Ableitung** übernehmen
- **Basisklasse:** Eine Klasse kann als Basis zur Entwicklung einer neuen Klasse dienen, ohne dass ihr Code geändert werden muss. Dazu wird die neue Klasse definiert und dabei angegeben, dass sie eine abgeleitete Klasse der Basisklasse ist.
- Alle öffentlichen Elemente der Basisklasse gehören auch zur neuen Klasse, ohne dass sie erneut deklariert werden müssen.
- Wiederverwendung des Codes
- Spezialisierung



Vererbung (2)

```
class Person {  
public:  
    string Name, Adresse, Telefon;  
};  
  
class Partner : public Person {  
public:  
    string Kto, BLZ;  
};  
  
class Mitarbeiter : public Partner {  
public:  
    string Krankenkasse;  
};  
  
class Kunde : public Partner {  
public:  
    string Lieferadresse;  
};  
  
class Lieferant : public Partner {  
public:  
    tOffenePosten *Rechnungen;  
};
```



Vererbung (3)

- Subtyp kann an die Stelle eines Basistyps treten:

```
Person person;
Mitarbeiter mitarbeiter;

person = mitarbeiter;      // ok
mitarbeiter = person;      // das mag der Compiler nicht
```

- Subklassen-Konstruktor 'called' zunächst Basisklassen-Konstruktor
 - Prinzip der kaskadierenden Konstruktoren
 - Bei den Destruktoren genau umgekehrt
 - Copy-Konstruktor wird nicht automatisch vererbt

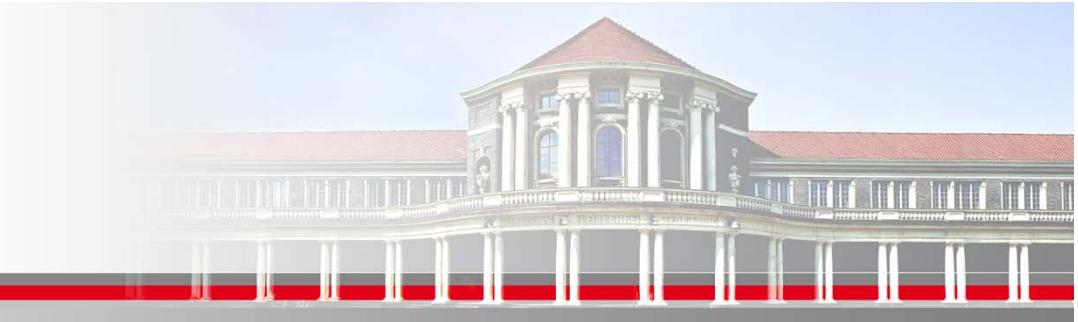


Methoden überschreiben

```
class tBasis
{
public:
    int TuWas(int a);
};

class tSpezialfall : public tBasis
{
public:
    int TuWas(int a);
};

int tSpezialfall::TuWas(int a)
{
    int altWert = tBasis::TuWas(a);
    ...
    return altWert;
}
```



Kapselung

```
class Basis {
private:
    int privat;
protected:
    int protect;
public:
    int publik;
};

class Abgelitten : public Basis {
    void zugriff()
    {
        a = privat; // Das gibt Ärger!
        a = protect; // Das funktioniert.
        a = publik; // Das funktioniert sowieso.
    }
};

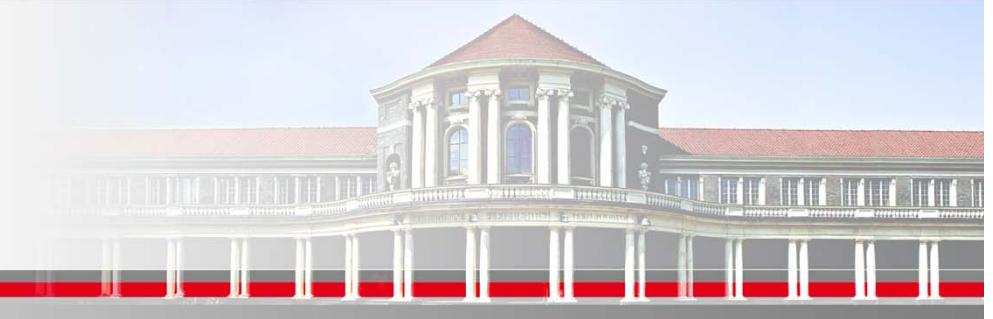
int main()
{
    Basis myVar;
    a = myVar.privat; // Das läuft natürlich nicht.
    a = myVar.protect; // Das geht auch nicht.
    a = myVar.publik; // Das funktioniert.
}
```



Templates (1)

- Templates unterstützen direkt *generic programming*
 - *Generic programming* = Datentypen sind Parameter in Deklarationen
 - So ähnlich wie formale Argumente in "normalen" Deklarationen später tatsächliche Daten (= Werte) aufnehmen
 - Definition der Parameter von Funktionen und Klassen erfolgt durch Datentypen
- Verwende Template Funktionen um gleiche Operationen für zu unterschiedliche Typen definieren
- Beispiel:

```
// gibt größten Parameterwert zurück
template <class T> T max(T a, T b)
{
    return a > b ? a : b ;
}
```



Templates (2)

```
void main()
{
    // max(int,int) is instantiated
    cout << "max(10, 15) = " << max(10, 15) << endl ;
    // max(char,char) is instantiated
    cout << "max('k', 's') = " << max('k', 's') << endl;
    cout << "max(10.1, 15.2) = " << max(10.1, 15.2) <<
endl;
}
```

- Compiler erkennt Type der Eingangsparameter
- Eine Instanz einer Funktion wird dementsprechend generiert



Template-Klassen (1)

- Eine typische Template-Klasse:

```
template <class T>
class CStack
{
public:
    CStack(int = 10) ;
    ~CStack() { delete [] m_pStackPtr ; }
    bool Push(const T& crItem);
    bool Pop(T& rResult) ;
    bool IsEmpty() const { return m_iTop == -1 ; }
    bool IsFull() const { return m_iTop == m_iSize - 1; }

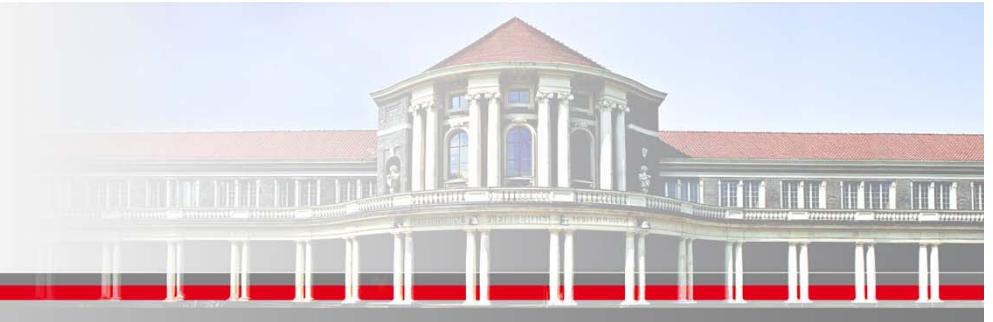
private:
    int m_iSize ; // Zähler für Anzahl Elemente auf Stack
    int m_iTop ;
    T* m_pStackPtr ;
} ;
```



Template-Klassen (2)

```
// Konstruktor; vordefinierte Größe (m_iSize) ist 10
template <class T>
CStack<T>::CStack(int iS)
{
    m_iSize = iS > 0 && iS < 1000 ? iS : 10 ;
    m_iTop = -1 ; // initialisiere Stack
    m_pStackPtr = new T[m_iSize] ;
}

// speichere einen Wert auf Stack
template <class T>
int CStack<T>::Push(const T& crItem)
{
    if ( !IsFull() )
    {
        m_pStackPtr[++m_iTop] = crItem ;
        return true ; // erfolgreich
    }
    return false ; // fehlgeschlagen
}
```

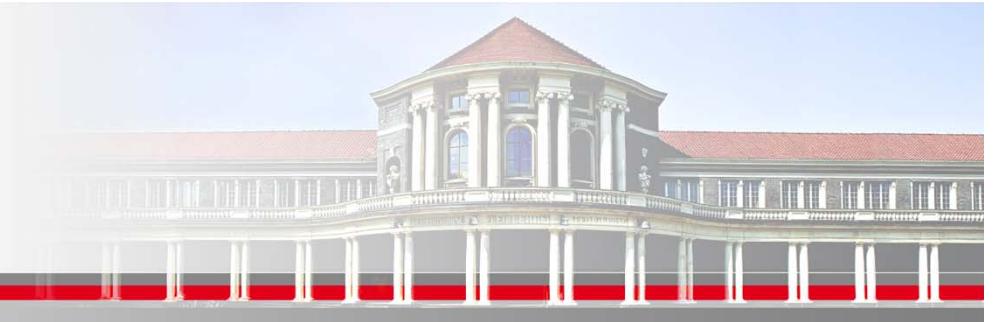


Template-Klassen (3)

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

void main()
{
    typedef CStack<float> FloatStackType ;
    typedef CStack<int> IntStackType ;

    FloatStackType cl_fs(5) ;
    float f_f = 1.1 ;
    while ( cl_fs.push(f_f) )      // neues Elements bis
    {                                // Stack voll ist
        cout << f_f << ' ' ;
        f_f += 1.1 ;
    }
}
```



Template-Klassen (4)

```
// schreibe alle Elemente von cl_fs nach stdout
while ( cl_fs.pop(f_f) )
    cout << f_f << ' ';

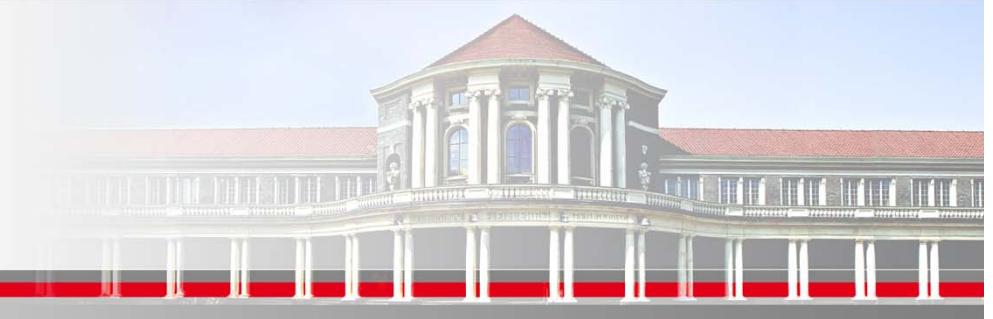
IntStackType cl_is;
int i_i = 1;
while ( cl_is.push(i_i) )
{
    cout << i_i << ' ';
    i_i += 1;
}

// schreibe alle Elemente von cl_is nach stdout
while ( cl_is.pop(i_i) )
    cout << i_i << ' ';
}
```



Template-Klassen (5)

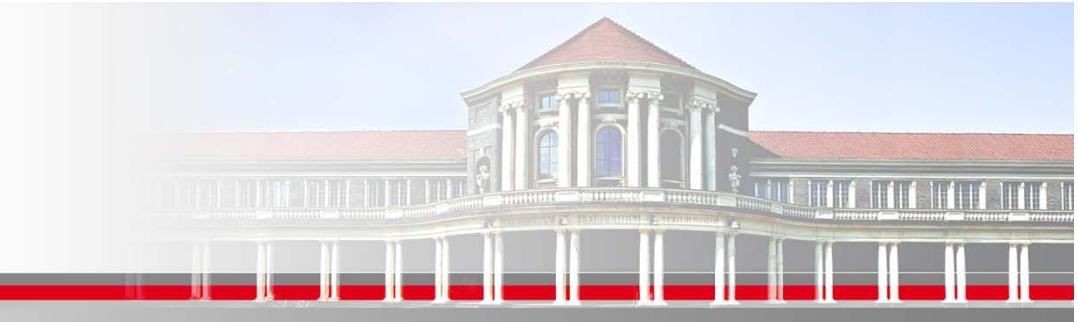
- Die Deklaration und Definition von *generic classes/functions* (d.h. Templates) gehört in *eine* Datei (nicht zwei)
- Organisiere Deklaration und Definition zweckmäßigerweise so:
 - Deklaration in einem Header-File (`.h`), Implementierung in einem Source-File (`.cpp`, `.hh` oder `.inl`) und binde die Source Dateien am Ende des Header-Files ein.
 - Achtung: Kompiliere nicht den `.cpp`-File !!!



Standard-Template-Library (STL) (1)

- Die Standard Template Library enthält viele effiziente Container, Algorithmen u.v.m., die für eigene Zwecke verwendet werden können
- Beispiel: dynamische Arrays

```
#include <vector>
...
vector<float> a; // default-Größe (meist 0)
vector<float> b(10); // 10 Elemente
```



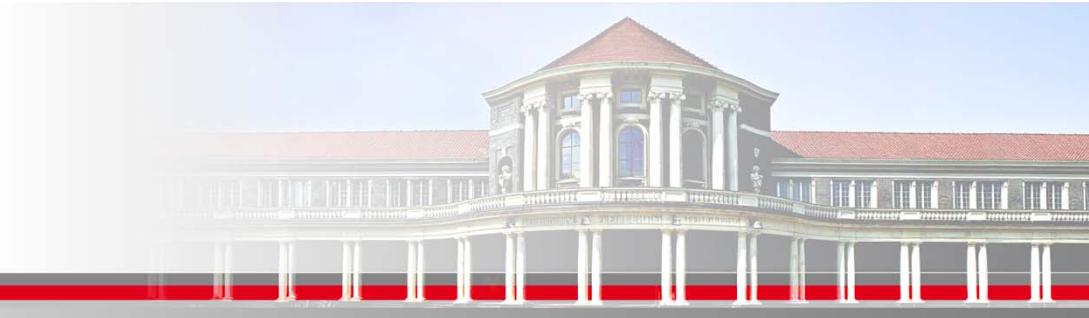
Standard-Template-Library (STL) (2)

- Verwendung wie bei herkömmlichen Arrays, zusätzlich z.B.:
 - Hinzufügen weiterer Elemente:

```
a.push_back(2.87f); // hinten
```

- Abfragen der Größe:

```
a.size();
```

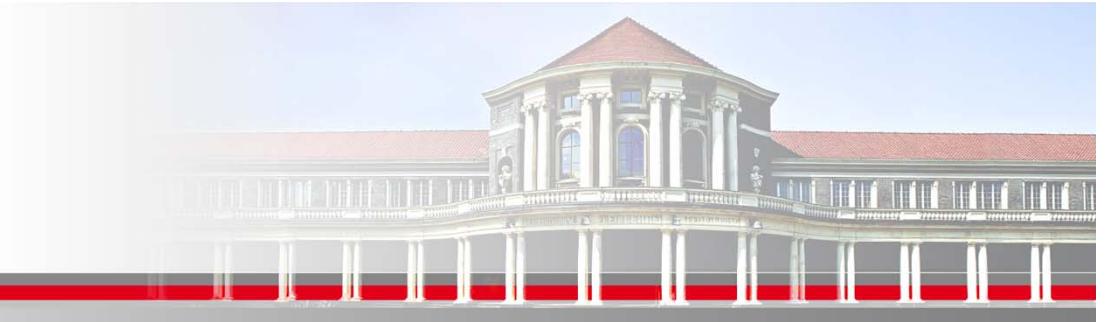


Standard-Template-Library (STL) (3)

- Eigene Datentypen können ebenso verwendet werden, z.B.:

```
#include "Vector2D.h"
#include <vector>

using namespace std;
...
vector<Vector2D> points;
points.push_back( Vector2D(1, 5) ); // anonyme Instanz
points.push_back( Vector2D(-3, 0) );
printf("%d ...", points.size());
```



Standard-Template-Library (STL) (4)

- Weitere nützliche STL-Komponenten
 - Container wie `map`, `list`, `stack`
 - Algorithmen wie `find()`, `sort()`, `min()`
 - Zeichenketten `string`