Weiterführende Programmiersprachen

C++ mit Java Vorkenntnissen

Dipl.-Ing. Thomas Marczinkowsky
Nauheimer Straße 83
70372 Stuttgart
Tel.: 0711-530 696 55

thomas.marczinkowsky@insystas.de

Überblick

- Vorlesung besteht aus zwei Teilen
  - C++ mit Java Vorkenntnissen
  - Die Programmiersprache C#
- Vorlesungsblock und Übungsblock
- Entwicklungsumgebung
  - Visual Studio 2019 Professional
  - Visual Studio 2019 Community

Prüfungsmodalitäten

- ca. 6 Programmieraufgaben, individuell
- Abgabe in ILIAS
  - Weiterführende Programmiersprachen
    - Kurs „C++ mit Java Vorkenntnissen“
    - Kurs „C# mit Java Vorkenntnissen“
- Erfolgreicher Abschluss
  - Bei mind. 50% der aufsummierten möglichen Punkte pro Programmiersprache
- Kontakt
  - thomas.marczinkowsky@insystas.de
  - Telefon: 0711-530 696 55
  - Mobil: 0163-65 116 11

Abstammungsgeschichte

- Programmiersprache C
  - Höhere Programmiersprache, prozedural
  - Strukturierter Ansatz
  - Dateien sind zu übersetzen (-> Compiler) und zu Binden (->Linker)
- Programmiersprache C++
  - Sprache C erweitert um Klassenkonzept
  - Generische Programmierung
  - Höhere Typsicherheit
- Programmiersprache Java
  - Vereinfachungen und Verbesserungen zu C++
  - Rein objektorientierte Sprache
  - Läuft auf „Virtueller Maschine“
- Programmiersprache C#
  - Alternative Entwicklung von Microsoft für .NET Framework
  - Konzepte von Java und C++
  - Schwerpunkt auf Laufzeiteffizienz

Entwicklung der Sprache C++

- 1979: Start der Entwicklung (Stroustrup)
- „C mit Klassen“
  - Klassen mit Datenkapselung
  - Strengeres Typsystem
- Compiler „cfront“ -> Cross Compiler
- 1982: Umbenennung in „C++“
- 1985: Erste Veröffentlichung von C++
- 1989: C++ Version 2.0
- 1998: Standardisierung: ISO/IEC 14882:1998
- 2003: Überarbeitung der Norm: ISO/IEC 14882:2003
- 2003/2005: Microsoft Erweiterungen wg CLI Kompatibilität
- 2011: wesentliche Erweiterungen der Sprache C++ (C++11)
- 2014: Anpassungen zu den Erweiterungen von 2011 (C++14)
- 2017: Zusätzliche neue Spracherweiterungen (C++17)
- 2020: momentan neueste Spezifikation (C++20)

Vorteile der Sprache C++

- Erzeugung hocheffizienten Codes
  - Native Code
- Sowohl maschinennahe als auch abstrakte Programmierung möglich
- Hohe Einsatz-Flexibilität
- Geeignet für große Projekte
- Weite Verbreitung

Nachteile der Sprache C++

- Historischer Ballast (-> Sprache C)
  - Präprozessor
  - Schwer verständliche Syntax
  - Compiler-spezifische Sprachanteile (wenige!)
  - Heterogener Ansatz: prozedural und objektorientiert
- Lange Einarbeitungszeiten
- Rückständige Standardbibliotheken

Das Hauptprogramm main() in C++

- Hauptprogramm ist nicht Teil einer Klasse
- Im gesamten Programm darf es nur ein main() geben
- Signatur:
  - **int main()**
  - **int main(int argNum, char* argArray[])**
    zur Übergabe von *argNum* Parametern aus der aufrufenden Kommandozeile, die im Feld *argArray* als Zeichenketten übergeben werden

Hauptprogramm

```
import java.io.*;
class ZweierPotenzen
{
        public static void main(String args[])
        {
                int zahl = 1;
                while (zahl <= 256)
                {
                        System.out.println("Zweierpotenz: " +
zahl);
                        zahl = zahl * 2;
                }
        }
}
```

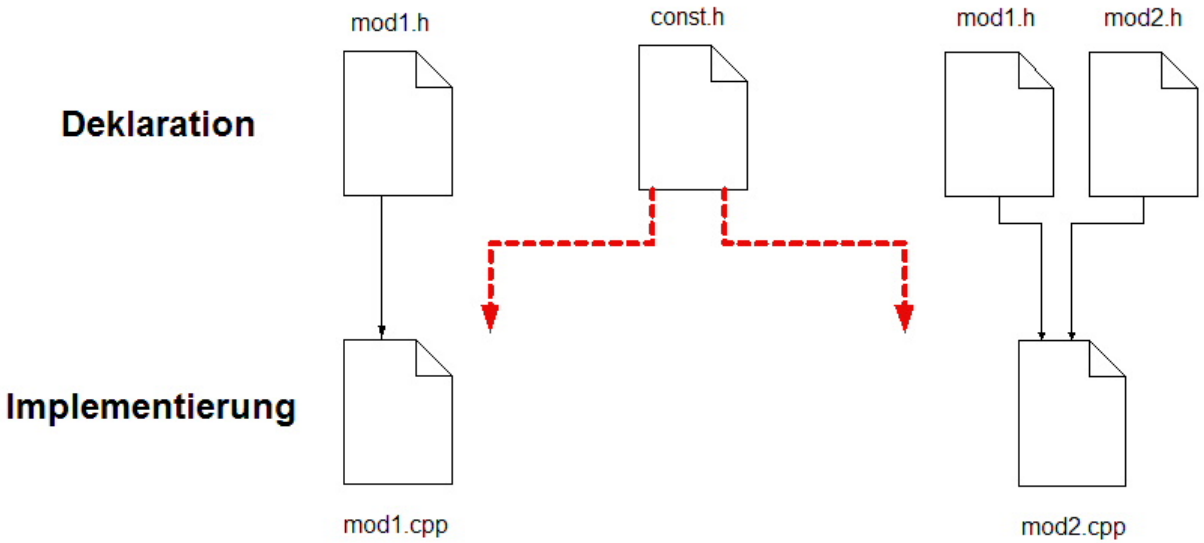
Java

```
#include <stdio.h>

int main(void) {
        int zahl = 1;

        while (zahl <= 256)
        {
                printf("Zweierpotenzen: %d\n", zahl);
                zahl = zahl * 2;
        }
        return 0;
}
```

C++

Dateiorganisation in C++

Modulkonzept



Dateiorganisation in C++

- Deklaration in einem Header-File
  - Deklaration einer Klasse, ihrer Methoden und Attribute
  - Endung ***.h** (seltener ***.hpp**)
  - Enthält leider auch private Methoden und Attribute einer Klasse (Kapselung?)
  - Enthält fast keinen Code für die Methoden (Ausnahme: inline)

Beispiel für ein Header-File

```
// Temperature.h

class Temperature
{
private:
      double celsius;
      double fahrenheit;
public:
      void SetCelsius(double value);
      void SetFahrenheit(double value);
      double GetCelsius() { return celsius; };
      double GetFahrenheit() { return fahrenheit; };
};
```

Das Implementierungs-File zur Klasse

- Implementierung von Methoden
  - Deklarierte Methoden müssen implementiert, d.h. mit Code gefüllt werden
  - Üblich: gleicher Filename wie Header-File
  - Endung: ***.cpp**
  - Muss Header „inkludieren“ (-> Präprozessor)

Beispiel für Implementierungs-File

```
// Temperature.cpp

#include “Temperature.h“

void Temperature::SetCelsius(double value)
{
    celsius = value;
    fahrenheit = 9.0 / 5.0 * celsius + 32.0;
}

void Temperature::SetFahrenheit(double value)
{
    fahrenheit = value;
    celsius = 5.0 / 9.0 * (fahrenheit – 32.0);
}
```

Hauptprogramm

```
// TempMain.cpp

#include “Temperature.h“
#include <iostream>

using namespace std;

int main()
{
    Temperature tempC;
    double tempValueC;
    cout << “Geben Sie eine Temperatur in Celsius ein: “ << endl;
    cin >> tempValueC;
    tempC.SetCelsius(tempValueC);
    cout << “Sie haben “ << tempC.GetCelsius() << “ C, das sind “
    << tempC.GetFahrenheit() << “ F eingegeben.\n“;
}
```

Präprozessor Anweisungen

- Präprozessor läuft vor dem Compiler
  - Reiner Textprozessor
    -> Code wird nicht interpretiert!
  - Sucht im File nach „#“ am Zeilenbeginn
  - Führt die Präprozessor Anweisung aus
- Die häufigsten Anwendungen
  - **#include** fügt ein File ein
  - **#define**
    - definiert Präprozessor-Variable
    - definiert Präprozessor Makro
  - **#ifdef/#ifndef** prüft, ob Präprozessor-Variable definiert ist
  - **#pragma**: Direktive für Compiler-spezifische Features

Schutz vor Mehrfach-Include

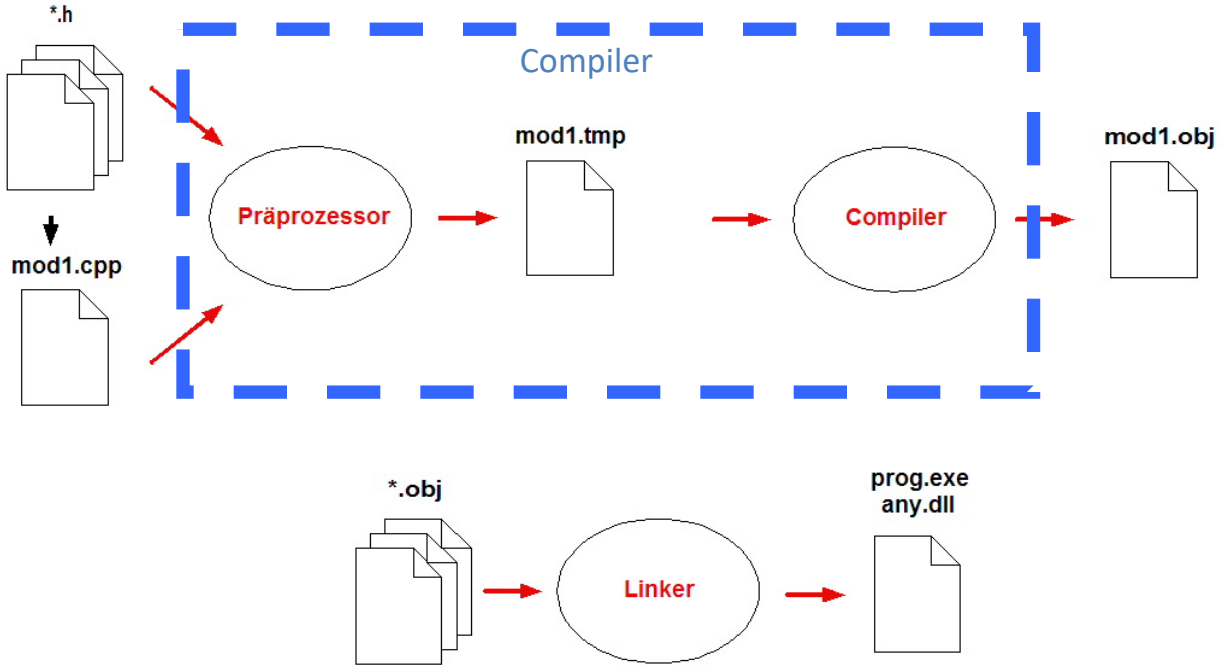
```
#ifndef _TEMPERATURE_H_   // prüft, ob schon included
                                                    // ab hier bedingter Teil
#define _TEMPERATURE_H_  // definiert Precomp.-Variable

// Temperature.h
class Temperature
{
...
}

#endif                                              // Ende des bedingten Teils
```

- Jedes Header-File so ausstatten!
- Alternative für Windows:
  - **#pragma once** // proprietär!

Übersetzen und Binden

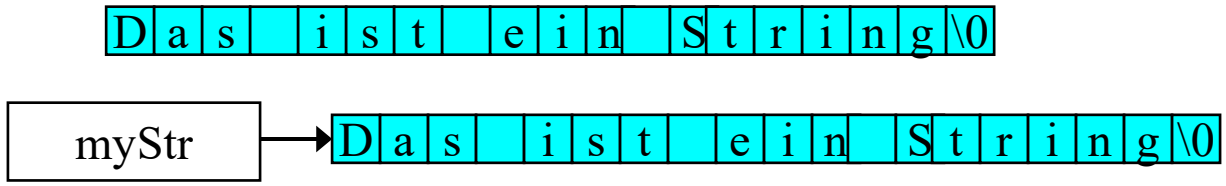
- Prinzipiell auch von Kommandozeile möglich
  - Verwendung mit Werkzeug: **make**
  - Wird im Rahmen der Vorlesung nicht behandelt
- Compilieren (Übersetzen)
  - Vor dem eigentlichen Compiler wird der *Präprozessor* aufgerufen
  - Vorgang für jedes Implementierungs-File einzeln notwendig
  - Compiler erzeugt Objekt-Files
- Linken (Binden)
  - Alle Object-Files werden zu ausführbarem Programm vereint
  - Linker kann auch *Dynamic Link Libraries (DLL)* erzeugen

Übersetzen und Binden



Definition von Strings als Arrays

- C++ erbt von C den Typ **char** als ASCII Zeichen
  - **char** hat die Länge 1 Byte
  - Java, C#: Unicode, d.h. **char** hat die Länge 2 Byte
- Zeichenkette = Array von char + '\0'
  - Literalkonstante **"Das ist ein String"**
  - String mit 18 Zeichen braucht 19 Byte zur internen Repräsentation
- Arrayvariable = Zeigervariable, Pointer
  - **char * myStr = "Das ist ein String";**



Lesen und Schreiben von Strings

- Zeichenketten in C++ sind nicht **immutable**

- Lesezugriff auf Zeichen mit **[ ]**

```
char c = myStr[0]; // c hat den Wert 'D'
```

- Schreibzugriff auf Zeichen mit **[ ]**

```
myStr[7] = 'k'; // nun: "Das istkeinString"
```

– Längenbestimmung: Zählen bis **'\0'**

```
int laenge = 0;
while (myStr[laenge] != '\0') laenge++;
```

Standardfunktionen aus <string.h>

- Längenermittlung: `int strlen(char[])`
  - `laenge = strlen(myStr);`
- Kopieren: `char strcpy(char* dest, char* src)`
  - Kopiert Inhalt von **src** nach **dest**
  - **src** muss null-terminiert sein
  - **dest** muss lang genug sein (Buffer Overflow)
  - `char strncpy(char* dest, char* src, int n)` kopiert maximal n Zeichen (sicherer)

Zusammenfügen von Strings

- **`char* strcat(char* dest, char* src)`**
  - Hängt **src** an **dest** an
  - **dest** kann leer sein, muss aber null-terminiert sein
  - Keine Sicherheit gegenüber Längenüberschreitung

```
char s1[80];                        // leerer Puffer für 79 Zeichen + End-0
strcpy(s1, "Diesen String"); // fülle Puffer mit Text
char s2[] = " kann man verlängern!"; // mit konst. String initialisiert
cout << strcat(s1, s2);    // Zusammenfügen
```

Formatierte Ausgabe

- Funktion **printf** aus der Sprache C
  - #include <stdio.h> notwendig
  - Formatiert und gibt danach auf dem Bildschirm aus
  - Signatur: `printf("Formatstring", par1, par2,…)`
  - Platzhalter: `%d, %5.3f, %c, %s…` nach Datentyp
- **sprintf** formatiert Daten zu Gesamtstring
  - Brauchbar zum Formatieren mit anschließender Ausgabe in GUI Elementen
  - Signatur: `sprintf(str, "Formatstring", par1, par2…)`

Ein- und Ausgabestream

- In Java: System.out, System.in
  - Mit Methoden `println(), read()`
- In C#, C++: `cin, cout`

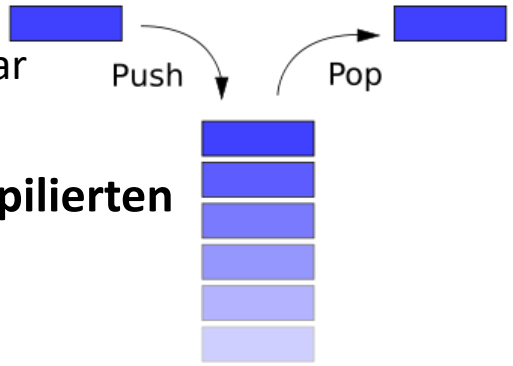
```
cout << "some words";
cin >> myVar;
```

- Voraussetzungen

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

Speicherverwaltung: Stack

- Stack: Stapelspeicher, häufige Datenstruktur
  - Nimmt eine beliebige Menge gleich langer Informationen auf
  - Gibt diese in umgekehrter Reihenfolge wieder zurück
  - Wird direkt von der Hardware unterstützt
    - + geringe Laufzeitkosten
    - - in der Regel nur begrenzt verfügbar
  - Operationen: **PUSH** und **POP**
  - **Achtung: Stacks werden vom compilierten Programm implizit verwendet!**
    - z.B. Parameterübergabe



Speicherverwaltung: Heap

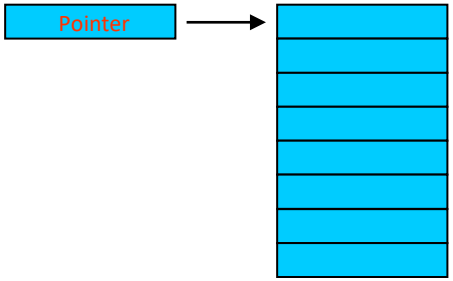
- Dynamischer Speicher
  - + im Prinzip beliebig verfügbar
  - - Fragmentierung
  - - Overhead für Verwaltung
- Wird zur Laufzeit vom Betriebssystem angefordert
  - **Java / C#**: new Operator
    - Freigabe: **Automatische Garbage Collection**!
  - **ISO-C**: malloc() / realloc()
    - Freigabe: free()
  - **ISO-C++**: new Operator
    - Freigabe: delete Operator

Objektinstanzierung in C++

- C++ gibt dem Programmierer die Freiheit, selbst zu entscheiden, in welchem Speicher die Instanz eines Objekt erstellt werden soll
- Instanzierung auf dem Stack
  - `MyClass foo;`
    Achtung: hier findet eine Instanzierung statt!
- Instanzierung auf dem Heap
  - `MyClass * pFoo = new MyClass(arg);`

Zeiger und Objektinstanzen

- C++ kann Zeiger (engl.: pointer) für Objektinstanzen anlegen
- Ein Zeiger ist eine Variable
- Eine Zeigervariable enthält eine Adresse
  - Zeigervariable ist 4 Byte lang
    (bei einem 32 Bit Betriebssystem)




```
int main() {
Temperature aussenTemperatur; // eine Instanz auf dem Stack
Temperature* pInnenTemperatur; // nur ein Zeiger, nichts drin
pInnenTemperatur = &aussenTemperatur; // "aliasing"
pInnenTemperatur = new Temperature; // neue Instanz auf Heap
}
```

Verwendung von Zeigern

- Dereferenzieren eines Zeigers
  - Heißt: Zugriff auf das Objekt selbst
  - **(*pInnenTemperatur).celsius**

    oder gleichbedeutend
  - **pInnenTemperatur->celsius**
  - **pInnenTemperatur->GetCelsius()**
- Zeiger sind „unsichere“ Referenzen
  - Zeigerarithmetik erlaubt
  - Zeigt nicht automatisch auf ein Objekt

void* Zeiger, NULL Zeiger

- Typenloser Zeiger: void*
  - darf mit Adressen/Zeigern beliebig gefüllt werden
  - entspricht in Java dem allg. Typ **object**
  - keine Typprüfung durch Compiler möglich!
- Leere Zeiger: 0 / 0x0 / NULL
  - Zeiger definiert auf NULL setzen
  - Erlaubt Erkennen ungültiger Zeiger

Objekte und Klassen (OOP)

- C++ verfügt zur OOP über ein Klassenkonzept
  - Identität: Instanz einer Klasse
  - Zustand: Attribute
  - Verhalten: Methoden / Operatoren
- Elemente objektorientierter Programmierung
  - Datenkapselung
  - Vererbung
  - Polymorphie

```
class CBeispiel
{
   int i;                          // Attribut
   void func();   // Methode
};

CBeispiel myObj;                   // Instanz einer Klasse
```

Datenkapselung

- Es existieren 3 *access specifier*
  - **public**
    Bezeichner kann von jeder Methode/Funktion verwendet werden
  - **protected**
    Bezeichner kann in Methoden der eigenen und abgeleiteter Klassen verwendet werden
  - **private**
    Bezeichner kann nur in der eigenen Klasse verwendet werden
- Default: private

Konstruktor und Destruktor

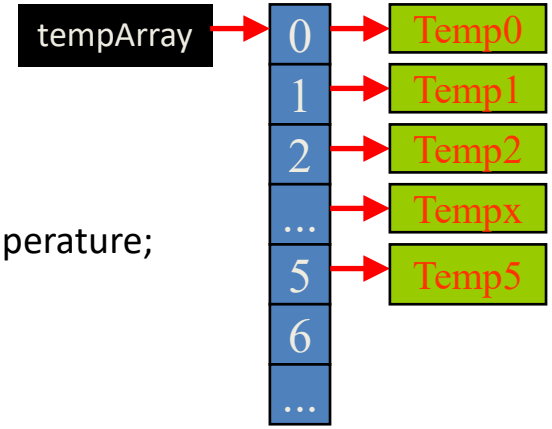
- Konstruktoren haben keinen Rückgabewert
  - Name ist identisch mit dem Klassenbezeichner
- Jede Klasse hat einen Default-Konstruktor
  - Keine Übergabeparameter
- Klasse kann weitere Konstruktoren haben
  - Unterscheiden sich durch den Parametersatz
  - Aufruf mit entsprechenden Parametern beim Anlegen auf Stack / Heap
- Klasse kann Destruktor haben
  - Namensgebung: **~Klasse()**
  - Automatischer Aufruf beim Ende der Lebensdauer des Objekts
    (z.B. manuelle Garbage Collection)

Beispiel für Konstruktor und Destruktor

```
// TemperatureField.h

#include "Temperature.h"

class TemperatureField
{
private:
        int validCount;                                  // Anzahl benutzter
Werte

public:
        Temperature * tempArray[100];      // Array mit lauter Zeigern,
                                                         // Anzahl 100 fix
        TemperatureField(int count);       // Konstruktor mit Parameter
        ~TemperatureField();               // Destruktor
};
```

Implementierung Konstruktor/Destruktor

```
TemperatureField::TemperatureField(int count)
{
        validCount = count;
        for (int i=0; i < count; i++)
                tempArray[i] = new Temperature;
}

TemperatureField::~TemperatureField()
{
        cout << "Destructor for TemperatureField called.\n";
        for (int i=0; i < validCount; i++)
                delete(tempArray[i]);
}
```



Testprogramm für Konstruktor/Destruktor

```
int main()
{
        // Array mit 6 gültigen Temperaturen anlegen
        TemperatureField theTemps(6);

        // TemperatureField mit Temperaturen füllen
        for (int k=0; k<6; k++)
          theTemps.tempArray[k]->SetCelsius(k*4.7);
        // Fahrenheit aus TemperatureField lesen
        for (int k=0; k<6; k++)  // ANSI/ISO: k neu zu definieren
          cout << theTemps.tempArray[k]->GetFahrenheit() << '\n';
} // Hier endet Lebenszeit von theTemps
```

Initialisierungen unter C++98

- Kein Standard-Initialisierungswert!
  - Pointer wie Variable – völlig zufällig!
  - Nur static-Variablen/-Attribute auf 0 initialisiert
- Initialisieren von Variablen
  - In der Definitionszeile durch = möglich
- Initialisieren von Attributen
  - Erst im Konstruktor möglich
- Initialisieren von statischen Attributen
  - Werden im Implementierungsfile initialisiert
  - Signatur: **<typ><klassenname>::<staticmember> = <wert>;**

Initialisieren eines Attributs einer Klasse (C++98)

- Zusätzlich möglich: Spezielle Syntax im Konstruktor
  - **Klasse::Klasse(int par1, char par2)**
    : attrName(value) {…
  - Das Attribut **attrName** erhält den Wert **value**
  - **value** kann symbolische/Literalkonstante sein
  - Ebenso möglich, den Initialwert aus den Parametern zu beziehen
  - Einzige Möglichkeit zur Initialisierung konstanter Attribute

Initialisierung von konstanten Attributen im Code

```
// TemperatureField.h
class TemperatureField
{
   private:
      int validCount;          // Anzahl benutzter Werte
      const int maxNoValues;   // maximale Zahl von Werten
...
```

```
// TemperatureField.cpp
TemperatureField::TemperatureField(int count)
: maxNoValues(100) // Festlegung der Konstanten auf 100
{
   validCount = count; // variables Attribut
...
```

Problemfälle bei der Initialisierung unter C++98

Unter C++98 ist es nicht möglich, ein C-Array als Attribut oder ein konstantes Heap-Array zu initialisieren!

```
class Example
{
private:
   int attr[4];
public:
   Example() : attr(...) {}
};

const int* pData = new const int[4];
```

attr kann nicht initialisiert werden!

Heap-Array kann nicht initialisiert werden!

Initialisierungen seit C++11

- C++98 Initialisierungen sind auch weiterhin möglich
  -> Abwärtskompatibilität des neuen Standards
- Attribute können nun auch an der Stelle ihrer Deklaration initialisiert werden
- C++11 erlaubt Initialisiererlisten für alle Initialisierungen

```
class Example
{
private:
   int attr[4]{1, 2, 3, 4};
public:
   Example() = default;
};

const int* pData = new const int[4]{1, 2, 3, 4};
```

Aufrufmethoden

- Call by value
  - Java: einziger Übergabemechanismus
  - C#, C++: Standardübergabe
- Call by reference
  - Java, C#: de facto Mechanismus bei Referenztypen
  - C#: für Werttypen: **ref, out**
  - C++: später eingeführte Notation
- Call by pointer
  - C++: spezielle Zeigervariante, historisch von C übernommen

Call by value

- Funktions-/Methodenaufruf mit Parametern
  - Kopien der Parameterwerte werden auf den Stack gelegt
  - In C++ werden alle Parameter, sogar ganze Objektinstanzen auf den Stack gelegt
  - Achtung: Arrayvariable ist Zeiger, Strings sind Arrays
  - Beispiel: **int x = MyFunction(var1, var2);**
- Wertänderungen an den Parametern innerhalb der Funktion/Methode verändern nicht das Original

Call by pointer

- Statt der ganzen Instanz wird nur der Zeiger übergeben
  - D.h. Zeiger wird kopiert und auf den Stack gelegt
  - Die Methode/Funktion dereferenziert den Zeiger und greift so auf die Originalinstanz zu
  - Änderungen bleiben nach dem Ende der Methode/Funktion am Original erhalten
  - Effizient, da nur der Zeiger (32 Bit) kopiert wird

Beispiel zu „Call by pointer“

```
float Angestellter::ErmittleEkST(EkStTabelle* pSteuerTab) {
        float steuer = pSteuerTab->LiesEkSt(gehalt);
        netto = gehalt - steuer;
        return steuer;
}
```

```
        EkStTabelle tabelle2004;
        ...

        cout << "Zu zahlende Steuer: " <<
         Erhardt_Paul.ErmittleEkST(&tabelle2004);
```

Call by reference

- „Call by pointer“ ist unschön, lästig
  - Durcheinander von Objekten und Zeigern
  - Ständiges Hantieren mit Adressoperator & und Dereferenzierungs-Operator *
- Handliche Syntax – gleiche Funktion!
  - Eine Referenz sieht aus wie ein Objekt
  - Funktioniert aber (mit Einschränkungen) wie ein Zeiger
  - Syntax: **int MyFunction (CMyClass & refPar);**

Beispiel zu „Call by reference“

```
float Angestellter::ErmittleEkST(EkStTabelle & steuerTab)
{
        float steuer = steuerTab.LiesEkSt(gehalt);
        netto = gehalt - steuer;
        return steuer;
}
```

'.' für Zugriff

Referenz-parameter

```
        EkStTabelle tabelle2004;
        ...

        cout << "Zu zahlende Steuer: " <<
         Erhardt_Paul.ErmittleEkST(tabelle2004);
```

Übergabe des Objekts selbst (als Referenz)

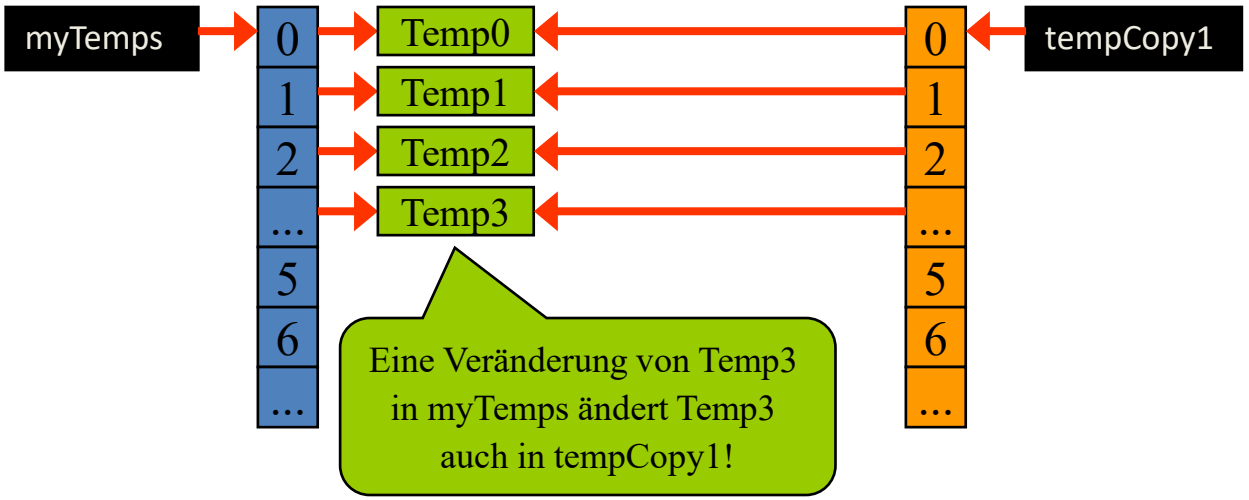
Kopieren von Objekten

- Java: Überladen der Methode clone()
  - Zuweisung kopiert nur die Referenz
- C++: Kopie durch Aufruf des Copy Konstruktors
  - **newObj(origObj)**
  - Vollständige Kopie aller Attributwerte
    aber: flache Kopie!
  - Für tiefe Kopie nötig: Überladen des Copy Konstruktors
    Klasse::Klasse(Klasse & orig);
- Weitere Möglichkeit: **newObj = origObj;**
  - Dazu notwendig: Überladen des Assignment Operators **operator=**

Beispiel für Objektkopie

```
// Beispiel für das Kopieren von Objekten
int main() {
          TemperatureField myTemps(4);
          // TemperatureField mit Temperaturen füllen
          for (int k=0; k<4; k++)
                    myTemps.tempArray[k]->SetCelsius(k*4.7);
          cout << "Original: "; myTemps.print();

          // eine Kopie des Objekts neu erstellen
          TemperatureField tempCopy1(myTemps); // Copy Constructor
          tempCopy1.tempArray[3]->SetCelsius(100);
          cout << "1. Kopie: "; tempCopy1.print();

          // Das Objekt durch Zuweisung kopieren
          TemperatureField tempCopy2(4);
          tempCopy2 = myTemps;          // Zuweisung = Assignment mit '='
          tempCopy2.tempArray[3]->SetCelsius(500);
          cout << "2. Kopie: "; tempCopy2.print();
}
```

Ergebnis der flachen Kopie



Copy Konstruktor für tiefe Kopie

```
// Copy Constructor

TemperatureField::TemperatureField(TemperatureField& orig)
{
          validCount = orig.validCount;
          for (int i=0; i < validCount; i++)
          {
             tempArray[i] = new Temperature;
             tempArray[i]->
                SetCelsius(orig.tempArray[i]->GetCelsius());
          }
          for (int i=validCount; i<100; i++)
                    tempArray[i] = NULL;
}
```

Assignment Operator

```
// Assignment Operator
TemperatureField& TemperatureField::operator=(TemperatureField& orig)
{
   // 1.: alten Inhalt aufräumen, wie Destruktor
   for (int i=0; i < validCount; i++)
   delete tempArray[i];
   // 2.: Inhalt des 'orig' kopieren, wie copy constructor
   validCount = orig.validCount;
   for (i=0; i < validCount; i++)
   {
      tempArray[i] = new Temperature();
      tempArray[i]->SetCelsius((orig.tempArray)[i]->GetCelsius());
   }
   // 3.: (Referenz auf) neues Objekt zurückgeben
   return *this;
}
```

Symbolische Konstanten

- Definition nicht-veränderlicher Variablen
  - **const float PI = 3.14159;**
- Deklaration durch Präprozessor
  - **#define PI 3.14159**
  - Präprozessor ersetzt Symbol durch Zahl
    -> rein textuell!
- Auch nicht-veränderliche Attribute möglich
  - **const int maxElements;**
  - Initialisierung durch **=** oder Init-Liste (seit C++11)

Datenstrukturen mit **struct**

- **struct** = „Vorstufe“ zu echter Klasse
  - Alle Attribute sind **public**
  - *Kann* auch Methoden haben (selten)
- Gruppiert zusammengehörende Daten zu einer strukturierten Einheit
- Verwendung in Klassen, Funktions- und Methodensignaturen, Datenbanken

Verwendungsbeispiel zu **struct**

```
// PersonalDatenstrukturen.h
struct persDatenT {
        char nachname[40];
        char vorname[20];
        int gebJahr;
        float grundgehalt;
        };
```

```
// Angestellter.h
#include "PersonalDatenstrukturen.h"

class Angestellter {
public:
        Angestellter(persDatenT d); // Konstruktor
};
```

Default Parameter

- Methodenaufruf mit weniger Parametern
  - Abweichung von strikter Methodensignatur
  - Hintere Parameter brauchen nicht belegt zu werden
  - Overloading auf die „faule Tour“
- Für optionale Parameter Defaultwerte
  - Ein Defaultwert wird im Funktions-/Methodenrumpf verwendet, wenn kein expliziter Wert übergeben wird

Beispiel für Default Parameter

```
class Angestellter {
        ...
        float bestimmeGehalt(float grundgehalt = 3000.0);
}
```

```
   Angestellter dr_duennwald;
   cout << "Dr. Dünnwald bekommt "
      << dr_duennwald.bestimmeGehalt(4500.) << " Euro\n";

   Angestellter strobel;
   cout << "Hr. Strobel bekommt "
      << strobel.bestimmeGehalt() // default 3000
eingesetzt
      << " Euro\n";
```

Operator Overloading

- Zweck: mathematische Schreibweise
  - Ausdrücke wie **t2 + t4** und **t5 == t6** für Objekte
  - dazu 40 Operatoren für Anwendungsklassen neu definierbar
- Definition: Methode **operatorXY**
  - Dabei steht XY für die Operation, wie +, ==
  - Methode wird klassenspezifisch implementiert

Operator Overloading - Implementierung

- Grundsätzlich 2 Möglichkeiten der Implementierung
  - Als *nicht-statische Methode* der Klasse
  - Als *Funktion*
- Einstellige Operatoren
  - Nicht-statische Methode ohne Parameter
  - Funktion mit einem Parameter
- Zweistellige Operatoren
  - Nicht-statische Methode mit einem Parameter
  - Funktion mit zwei Parameter

Überladen einstelliger Operatoren

- Methode

```
class Cxy {
          ...
          Cxy
operator++();
          ...
}
```

- Funktion

```
class Cxy {
          ...
}

Cxy operator++(Cxy&
arg);
```

Überladen zweistelliger Operatoren

- Methode

```
class Cxy {
          ...
          Cxy
operator+(Cxy&);
          ...
}
```

- Funktion

```
class Cxy {
          ...
}

Cxy
operator+(Cxy& arg1,
          Cxy& arg2);
```

Beispiele überladener Operatoren

```
// Temperature.h
        bool operator== (Temperature& other);
        Temperature operator+ (Temperature& other);
```

```
// Temperature.cpp
bool Temperature::operator== (Temperature& other) {
        return (celsius == other.celsius);
}
Temperature Temperature::operator+ (Temperature& other){
// gib neues Temperaturobjekt mit Summenwert zurück
        return Temperature (celsius + other.celsius); }
```

```
int main() { Temperature t1, t2, t3;
        if (t1 == t2)
                cout << "Temperaturen sind gleich.\n";
        t3 = t1 + t2;
```

Vererbung

- Vererben von Attributen und Methoden
  - Angabe der Elternklasse nach ":"
    class ChildClass : public ParentClass {…
  - Child-Objekt erbt Definition und Implementierung der *Methoden* der Parent-Klasse
  - Child-Objekt enthält alle *Attribute* der Parent-Klasse
  - Parent-Objekt wird sozusagen in Child-Objekt „eingebaut"

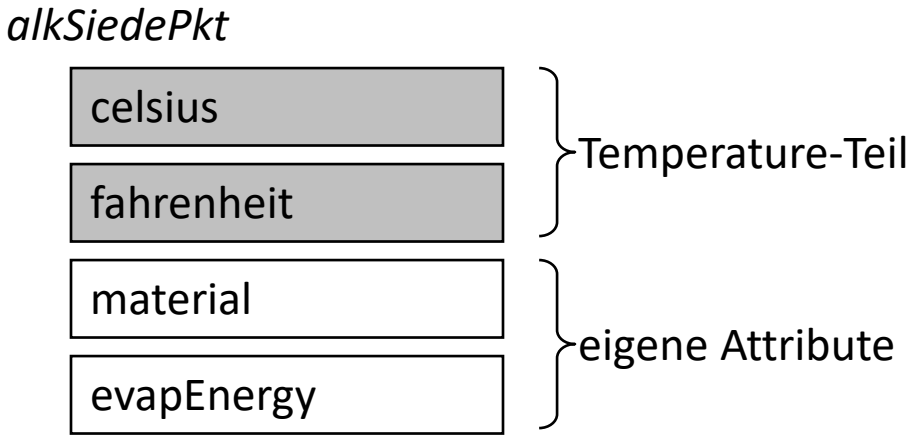
Beispiel für Vererbung

- Siedepunkt ist eine Temperatur
- Enthält Stoffbezeichnung und Verdampfungswärme in J/g

```
class BoilingPoint : public Temperature {
        float evapEnergy;
        char material[40];
public:
        void SetMaterial(char* mat);
        void SetEvapEnergy(float energy);
};
```

Zugang zu Attr./Methoden der Elternklasse

Innerer Aufbau bei Vererbung



Anwendung der Vererbung

```
#include "BoilingPoint.h"

int main()
{
    BoilingPoint alkSiedePkt;
    alkSiedePkt.SetMaterial("Ethanol");
    alkSiedePkt.SetEvapEnergy(858
    alkSiedePkt.SetCelsius(78.3);

    BoilingPoint wasserSiedePkt(100., "Water", 2258.);
}
```

Einsatz Default-Konstruktor

Spezieller Konstruktor

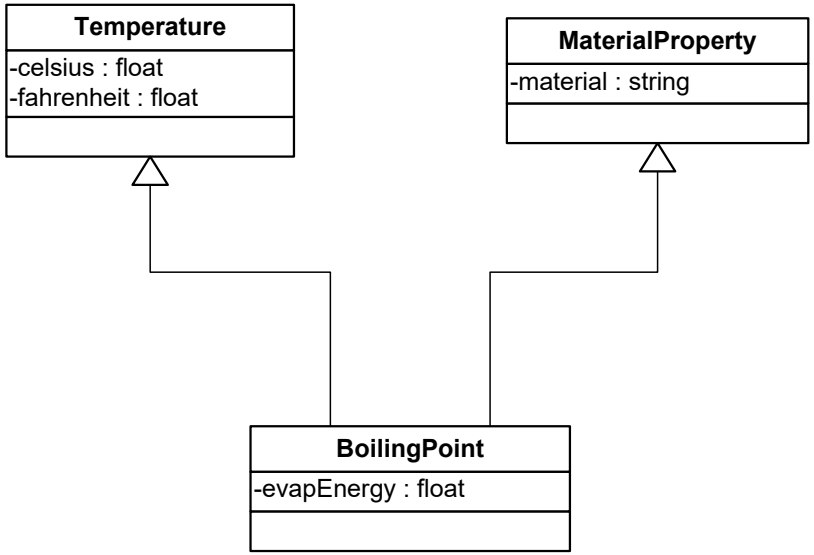
Konstruktoren bei Vererbung

```
#include "BoilingPoint.h"
#include <string.h>

BoilingPoint::BoilingPoint() {
        material[0] = 0; // leerer String
        evapEnergy = 0.;
}

BoilingPoint::BoilingPoint(float celsius,
                           char* mat, float energ
                           : Temperature(celsius){
        strcpy(material, mat); // String kopieren
        evapEnergy = energy;
}
```

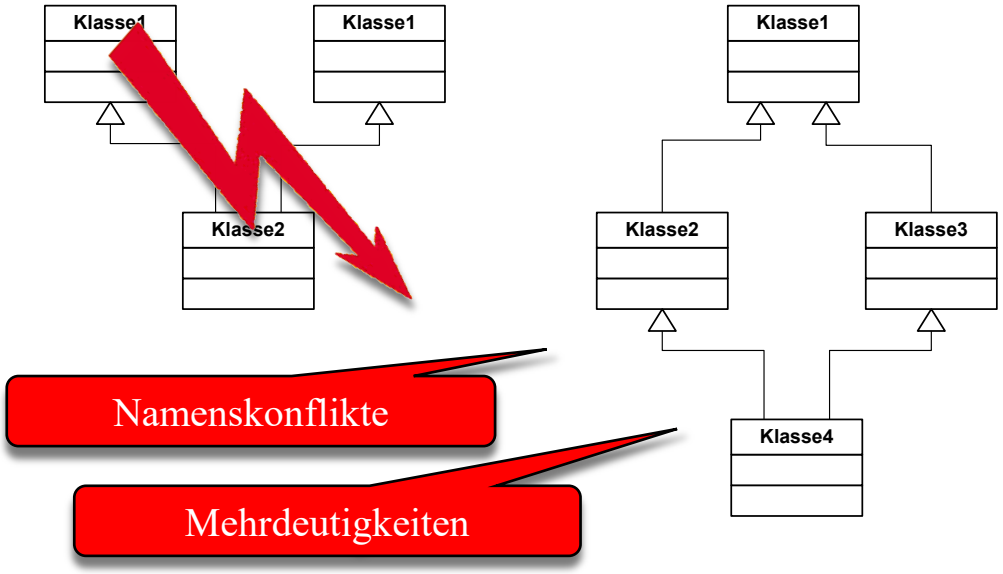
Impliziter Aufruf des Default-Konstruktors von Temperature

Aufruf eines bestimmten Konstruktors von Temperature

Mehrfach-Vererbung



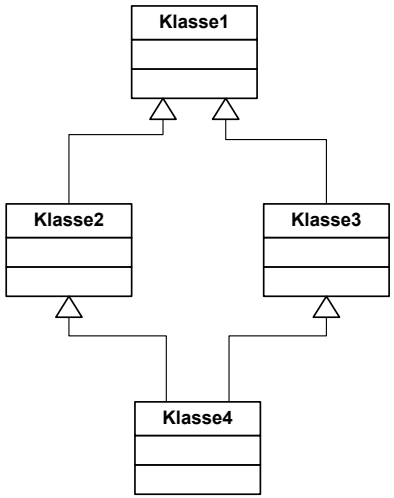
Probleme der Mehrfach-Vererbung



Direkte Basisklasse

Diamond-Problem

Das Diamond-Problem



- Mehrdeutigkeiten
  - Lösung 1: Scope Operator **::**

```
Klasse4 myClass;
…
myClass::Klasse2.SetCelsius(3.2)
;
```

Nachteil: doppelter Attribut-Bereich

  - Lösung 2: Virtuelle Basisklasse

```
Klasse2 : virtual Klasse1 {…};
Klasse3 : virtual Klasse1 {…};
```

Definition einer Mehrfach-Vererbung

```
// BoilingPoint.h

#include "Temperature.h"
#include "MaterialProperty.h"

class BoilingPoint : public Temperature,
                     public MaterialProperty {
    float evapEnergy;
public:
    void SetEvapEnergy(float energy);
    BoilingPoint();
    BoilingPoint(float celsius,char* mat,float energy);
};
```

Konstruktoren bei Mehrfach-Vererbung

```
// BoilingPoint.h

#include "Temperature.h"
#include "MaterialProperty.h"

class BoilingPoint : public Temperature,
                     public MaterialProperty {
    float evapEnergy;
public:
    void SetEvapEnergy(float energy);
    BoilingPoint();
    BoilingPoint(float celsius,char* mat,float energy)
        : Temperature(celsius),
          MaterialProperty(mat);
};
```

Virtuelle Methoden, Polymorphie

- Kennzeichnung durch virtual
  - Definition in der Basisklasse
- Erneute Definition in den speziellen Klassen
  - Mit gleicher Signatur und gleichem Rückgabewert
- Default-Implementierung
  - In Basisklasse: Standardbehandlung
- Implementierung wird überschrieben
  - In Child Klasse: spezielle Behandlung

Vergleich mit dem Überladen von Methoden

Überladen

- Art und/oder Anzahl der Argumente muss sich unterscheiden
- **Early Binding**:
  schon der Compiler erkennt an Hand der Argumentliste, welche Methode gemeint ist

Überschreiben (Polymorphie)

- Argumentliste und Rückgabewert sind gleich
- **Late Binding**:
  erst zur Laufzeit wird an Hand des Objekttyps entschieden, welche Methode aufgerufen wird

  => Laufzeit Overhead

Bedeutung virtueller Methoden

- Sammlung unterschiedlicher Objekte
  - Array of pointers auf Basisklasse
- Aufruf einer polymorphen Methode
  - Spezialbehandlung der Child Objekte
- Richtige Methode wird nur aufgerufen, wenn Methode der Basisklasse als **virtual** deklariert wird
  - Polymorphie durch **late binding**
  - Destruktor einer Basisklasse sollte virtuell deklariert werden!

Definition einer virtuellen Methode

```
class Bediensteter {
        float netto; float brutto; float ekst;
public:
        virtual float BestimmeNetto(float gehalt);
};
```

```
class Beamter : public Bediensteter {
public:
        float BestimmeNetto(float gehalt);
};
```

```
class Angestellter: public Bediensteter {
public:
        float BestimmeNetto(float gehalt);
};
```

Abstrakte Klassen

- Klassen, die nicht instanzierbar sind
  - In C++: „Krücke“ gebastelt
  - Es gibt **kein** Schlüsselwort **abstract**!
- „Rein virtuelle Methode“:
  - Methode, die nicht implementiert ist
  - Implementierung erfolgt in den Child Klassen
  - Signatur: `virtual int MyMethod() = 0;`
- Abstrakte Klasse:
  - Klasse mit mindestens einer **rein virtuellen** Methode

Abstrakte Klasse „Bediensteter“

```
class Bediensteter {
        float netto; float brutto; float ekst;

public:
        virtual float BestimmeNetto(float gehalt)
                                                    =
0;
};
```

Dies allein kennzeichnet die 'pure virtual function' und macht die Klasse abstrakt!

Typumwandlungskonstruktoren (Conversion Constructor)

- Konstruktoren mit einem Parameter definieren auch eine **implizierte Konvertierung**!
  - Kann eine sinnvolle Möglichkeit sein:
    z.B. Komplexe Zahl mit einem int Wert initialisieren
    ```
    complex z = 2; // initialisiere z mit complex(2)
    ```
  - Ist aber auch oft nicht erwünscht!
- Implizite Konvertierung kann unterdrückt werden durch Verwendung des Schlüsselworts **explicit** bei der Deklaration des Konstruktors

Beispiel zu Conversion Constructor

```
const unsigned int maxFeldLaenge = 50;

class Feld {
              int m_feld[maxFeldLaenge];
              unsigned int m_feldLaenge;
public:
              Feld(int groesse) {
                            for (int i = 0; i < groesse; i++) m_feld[i] = 0;
                            m_feldLaenge = groesse;
              }
};

int main() {
              Feld x(10);  // expliziter Aufruf
              x = 5;       // impliziter Aufruf
              Feld y = 10; // implizierter Aufruf
}
```

```
...
public:
              explicit Feld(int groesse) {
...
```

Funktionszeiger

- Bisher: Zeiger auf Daten
  - `MyClass *pThisClass;`
- Neu: Zeiger auf Funktionen
  - Zeiger auf einen Codeteil des Programms
  - Bei der Deklaration muss ‚geklammert' werden:
    `double (*pf)(double, double);`
    Bedeutung: Zeiger auf eine Funktion mit zwei Argumenten vom Typ double und einem Rückgabewert double
- Aufruf einer Funktion über einen Zeiger durch Dereferenzieren:
  - `double wert = (*pf)(1.5, 2.7);`

Beispiel zu Funktionszeigern

```
double square(double); // Definition an anderer Stelle
...

void something() {
        double (*pf)(double);
        pf = &square;
        ...

        double wert = (*pf)(2.7);
        ...
}
```

Dynamic Link Library (DLL)

- Unter dem Betriebssystem Microsoft Windows verwendete dynamische Bibliothek
- Format: Portable Executable (EXE)
- DLLs können enthalten
  - Programmcode
  - Daten
  - Ressourcen
- Windows-Dateiendung:
  - *.dll
  - *.ocx (ActiveX Steuerelemente)
  - *.ico (Icons)

Zweck von Dynamic Link Libraries

- Reduzierung von Speicherplatz auf Festplatte und im Hauptspeicher
  - Programmcode, der von mehr als einer Anwendung benötigt werden könnte, wird in einer DLL gespeichert
  - DLLs werden im Hauptspeicher nur einmal geladen, auch wenn mehrere Anwendungen darauf zugreifen

Vorteile und Schwächen von DLLs

- Vorteile
  - Effizienter Umgang mit Speicher-Ressource
  - Bei Verbesserung des Programmcodes müssen nicht alle Anwendungen geändert werden
  - Ganze Softwarepakete können durch die Aktualisierung der DLL auf den neuesten Stand gebracht werden
- Schwächen
  - DLL-Konflikt:
    Mehrere Anwendungen benötigen verschiedene Versionen der gleichen DLL
    => kann zu Installationsproblemen führen
  - Behebung des DLL-Konflikts durch Kopieren der richtigen Version in das Programmverzeichnis der Anwendung
    aber: damit wird Effizienz (zumindest teilweise) wieder aufgegeben
  - Microsoft Windows .NET umgeht dieses Problem durch ein Konzept zur gleichzeitigen Existenz von mehreren Versionen einer Bibliothek

Das Laden von DLLs bei einem Programmstart

- Jedes Windows Programm enthält eine **Import-Table**
  - *Import-Table* enthält alle *Ordnungszahlen* der DLL-Befehle, die vom Programm benutzt werden
- Weitere Schritte des **Loaders** (Betriebssystem)
  - Alle fehlenden DLLs, die vom Programm verwendet werden, werden in den Hauptspeicher geladen
  - In die Import-Table werden die *Einsprungadressen* der einzelnen Befehle eingetragen (Funktionszeiger!)

Datei Aufbau einer DLL

- DLL enthält ebenso wie normale ausführbare Programme einen *Header*, in dem aber das **IMAGE_FILE_DLL** Bit gesetzt ist
- DLLs enthalten eine **Export-Table**
  - *Export-Table* enthält alle Namen der Funktionen und Variablen, die eine DLL an externe Software zur Verfügung stellt
  - Auch normale ausführbare Programme enthalten eine *Export-Table*, die aber nur selten benutzt wird

Laden und Entladen einer DLL

- Betriebssystem ist verantwortlich für das Laden und Entladen der verwendeten DLLs
- Dazu wird ein **Instanzenzähler** vorgehalten
  - Bei der ersten Verwendung wird die DLL in den Hauptspeicher geladen und der *Instanzenzähler* auf den Wert „1“ initialisiert
  - Bei jeder weiteren Verwendung durch andere Anwendungen wird der *Instanzenzähler* inkrementiert
  - Wird eine Anwendung terminiert, so werden die *Instanzenzähler* aller von ihr verwendeten DLLs dekrementiert
  - Ein Instanzenzählerwert „0“ bedeutet, dass die DLL zur Zeit nicht mehr benötigt wird
- Bei Speicherbedarf entlädt ggfs. das Betriebssystem DLLs, deren *Instanzenzähler* auf dem Wert „0“ stehen

Erstellen einer DLL

- Die DLL-Schnittstelle wird mit Hilfe der Export-Funktion **__declspec(dllexport)** definiert.
  - Wird in **windows.h** zur Verfügung gestellt
- *Compiler* und *Binder* erzeugen
  - Die DLL mit der Endung *.DLL
  - Eine **LIB-Datei** (*.LIB), die in der konsumierenden Anwendung mit eingebunden werden muss
    - Entspricht in etwa einer Header-Datei beim Compiler

Beispiel für eine DLL

```
#include <windows.h>

#define DLL_EXP extern "C" __declspec(dllexport)

// Zu exportierende Funktion

DLL_EXP double square (double arg)
{
        return arg * arg;
}
```

Verwenden von DLLs in einer Anwendung

- Die Funktionen, die man in der Anwendung verwenden möchte, werden mit der Funktion **__declspec(dllimport)** importiert
- Beim Binden muss die zur DLL gehörende **LIB-Datei** mit eingebunden werden

```
#include <windows.h>
#include <stdio.h>

// Importieren der DLL Funktion
extern "C" __declspec(dllimport) double square(double);

int main()
{
        // Abrufen der externen Funktion
        double result = square(2.7);
        printf("Das Ergebnis ist: %f\n", result);
        return 0;
}
```

Generische Programmierung

- Parametrisierte Klassen und Funktionen
  - Programmierung mit Typen als Parametern
  - Klassen und Funktionen für einen „beliebigen“ Typ
  - Compiler generiert passende Version automatisch
    => **Early Binding**!
- Signatur:

```
template <Template-Parameterliste> Deklaration
```

- C++ Standardbibliothek STL verwendet Templates Konzept intensiv

Parametrisierte Klassen

- Problem: Verwaltung von Elementen unterschiedlichen Typs
  - Möglichkeit 1:
    Verwaltung über Zeiger und explizite Konvertierung (Casting)
    - Nachteil: keine Typsicherheit
  - Möglichkeit 2:
    Definition typspezifischer Klassen
    - Nachteil: Redundante Kodierung fast identischen Codes
  - Möglichkeit 3:
    Überladen von Methoden
    - Nachteil: auch hier redundante Kodierung
- Alternative: Realisierung mit Template Klasse
  - Für alle Typen gilt das gleiche Codesegment
  - Compiler generiert nur die tatsächlich benötigten Varianten
  - Achtung: keine Laufzeitverluste, da *Early Binding*!

Parametrisierte Klassen

Beispiel: Überladen

```
class String {
   struct Srep;
   Srep *rep;
public:
   String();
   String(const char*);
   String(const wchar_t*);
};
```

Alternative: Template Klasse

```
template <class C>
class String {
   struct Srep;
   Srep *rep;
public:
   String();
   String(const C*);
   …
   C read(int i) const;
};
```

Template-Parameter

- Ein Template kann mehrere Parameter haben
  - `template <class T, T stdWert> class Cont {/* … */};`
    - Ein Template-Parameter kann bei der Definition nachfolgender Parameter verwendet werden
  - `template <class T, int i> class Buffer {/* … */};`
    - Int-Argumente sind z.B. sinnvoll, um Größen oder Grenzen zu übergeben
    - Ein Template-Parameter, der kein Typ ist, gilt innerhalb der Template-Klasse als eine Konstante
    - Ein int-Argument muss eine Konstante sein!
      - `Buffer<int, i> myBuf; // Fehler!!`

Parametrisierte Funktionen

- Ebenso wie *Klassen* lassen sich auch *Funktionen* parametrisieren:

```
int mittel(int* const x, int
   n)
{
   int summe = x[0];
   for (int i=0;i<n;i++) {
           summe += x[i];
           return summe/n;
}
```

```
template <class P>
P mittel(P* const x, int n)
{
   P summe = x[0];
   for (int i=0;i<n;i++) {
           summe += x[i];
           return summe/n;
}
```

Aufruf parametrisierter Funktionen

- *Explizites Spezifizieren* der Template-Parameter
  - `cout << mittel<int>(a, 5) << endl;`
- Ohne Spezifikation der Template-Parameter
  - `cout << mittel(a, 5) << endl;`
  - Der Compiler versucht hier, an Hand der übergebenen Argumente die Template-Parameter abzuleiten
- Die Funktion **mittel** kann auch für selbst definierte Klassen aufgerufen werden
  - Achtung:
    Die im Funktionsrumpf verwendeten *Operationen* und *Methoden* müssen sinnvoll definiert sein.
    Dies gilt besonders für
    - den *Copy Konstruktor*
    - die Operatoren (*Operator Overloading!, siehe auch STL Bibliothek*)

Standardbibliothek STL

- Eigener Namensbereich: **std**
- Bereiche der Standardbibliothek
  - Ein-/Ausgabe **cout / cin**
  - Datentyp **string**
  - Container
    - Vektoren, Listen, Maps…
  - Algorithmen
  - Mathematische Berechnung
    - Komplexe Zahlen, Vektorarithmetik

STL: Der Datentyp **string**

- #include <string>
- Basisklasse:

```
template<class Ch, class Tr=char_traits<Ch>, class
=Allocator<Ch> >
class std::basic_string {
public:
    // …
}
typedef basic_string<char> string;
```

- Konkatenation durch + Operator
  - `cout << s1 + s2;`
- Zuweisung durch = Operator
- Konvertieren zu C-String: `c_str()`
- Diverse zusätzlich Methoden:

  `copy(), compare(), toupper(), insert()`

STL: Standardcontainer

- 2 Arten von Containern:
  - Sequentielle Container
    - Verhalten sich alle wie **vector**
  - Assoziative Container
    - Bieten zusätzlich Elementzugriff über *Schlüssel*
- Container sind Klassen, deren Hauptaufgabe die *Verwaltung von Objekten* ist
  - **Vorteil**:
    im Gegensatz zu Feldern haben Container eine variable Größe mit *dynamischer Speicherverwaltung*
- Bei Containern handelt es sich um **Template-Klassen**

STL: Liste der Standardcontainer

Containerklasse	Beschreibung
vector<T>	Ein variabel großer Vektor
list<T>	Eine doppelt verkettete Liste
queue<T>	Eine Queue
stack<T>	Ein Stack
deque<T>	Eine „double ended queue“
priority_queue<T>	Eine nach dem Wert sortierte Queue
set<T>	Eine Menge
multiset<T>	Eine Menge, in der ein Wert mehrfach vorhanden sein darf
map<key, wert>	Ein assoziatives Feld
multimap<key, wert>	Eine Map, in der ein Schlüssel mehrfach enthalten sein kann

STL: Hauptoperationen der Standardcontainer

Name	Beschreibung
value_type	Datentyp der Elemente
iterator	Abstrakter Zeiger
key_type	Datentyp des Schlüssels
begin()	Iterator auf das erste Element
end()	Zeigt eins hinter das letzte Element
push_back()	Element am Ende anfügen
pop_back()	Letztes Element entfernen
insert(p, x)	Fügt x vor p ein
erase(p)	Löscht Element bei p
clear()	Löscht alle Elemente

STL: Iteratoren

- **Iteratoren** sind Zeiger, mit denen man durch die Elemente einer Liste oder Menge *iterieren* kann
- Unterschied zu *Index* bzw. *Schlüssel*
  - Zugriff auf Element ohne Kenntnis der Datenstruktur
  - Gültig nur für genau eine Datenstruktur
  - Keine Serialisierung möglich

STL: Beispiel zu Iteratoren

```
#include <list>
Using namespace std;

class complex;

int main()
{
          list<complex> myList;
          list::iterator it;
          ...

          for (it = myList.begin(); it != myList.end(); it++)
          {
                    ...
          }

          return 0;
}
```

STL: Algorithmen

- Für häufig wiederkehrende Aufgaben in Zusammenhang mit Containern stellt die Standardbibliothek Algorithmen zur Verfügung
- Algorithmen der STL folgen dem *Sequenzkonzept*:
  - Eine **Sequenz** wird repräsentiert durch ein paar von *Iteratoren*, wobei der erste Iterator auf das erste Element der Sequenz zeigt und der zweite Iterator hinter das letzte Element der Sequenz

STL: Algorithmen und Prädikate

- Eine Funktion, die das Verhalten eines Algorithmus steuert, heißt **Prädikat**

```
bool gt_42(const pair<const string, int> & r)
{
        return r.second > 42;
}

void f(map<string, int> & m)
{
        typedef map<string, int>::const_iterator MI;
        MI i = find_if(m.begin(), m.end(), gt_42);

        ...
}
```

STL: Ausgewählte Standardalgorithmen

Name	Beschreibung
for_each()	Führt Funktion für jedes Element aus
find()	Findet erstes Auftreten des Arguments
find_if()	Findet erste Übereinstimmung des Prädikats
count()	Ermittelt die Häufigkeit des Elements
count_if()	Zählt Übereinstimmungen des Prädikats
replace()	Ersetzt Element durch neuen Wert
replace_if()	Ersetzt Element, das mit Prädikat übereinstimmt, durch neuen Wert
copy()	Kopiert Elemente
unique_copy()	Kopiert Elemente, die keine Duplikate sind
sort()	Sortiert Elemente
merge()	Mischt sortierte Sequenzen