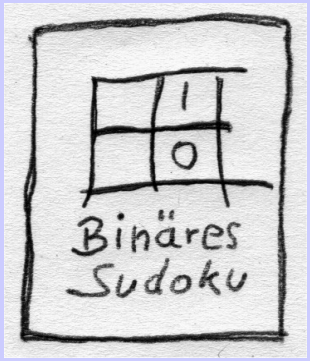
hochschule mannheim

Jörn Fischer
j.fischer@hs-mannheim.de

Willkommen zur Vorlesung

Informatik 2 für Mechatroniker



Überblick

Inhalt

1 - Organisatorisches

1 - Organisatorisches

Kapitel 1 - Organisatorisches

- Wie läufts ab?
- Prüfungsvoraussetzung
- Fragen an mich

1 - Organisatorisches

Wie läufts ab?

- Vorlesungsfolien finden Sie auf meiner Homepage:

  https://services.informatik.hs-mannheim.de/~fischer/lehre.html

- Übungen sollen auch zu Hause programmiert werden!
- Gerne können Übungen in 2'er oder 3'er Gruppen gelöst werden, testiert wird aber jeder Einzelne.

1 - Organisatorisches

Prüfungsvorraussetzung

- Die Aufgaben werden mit Punkten bewertet. Für richtig gelöste Aufgabenblätter gibt es 100 Punkte. Im Durchschnitt müssen 80 Punkte erreicht werden, um zur Prüfung zugelassen zu werden.
- Kommentare in den Programmen sind in englischer Sprache zu verfassen! Kommentare in anderen Sprachen (Deutsch, Türkisch, Französisch etc.) führen zu Punktabzug.
- Wenn Sie die Klausur mitschreiben möchten, müssen Sie sich rechtzeitig beim Prüfungsamt anmelden!
- Falls Sie sich nicht anmelden können, schreiben Sie bitte rechtzeitig eine Mail an das Prüfungsamt mit CC an mich!

1 - Organisatorisches

Fragen…

- Fragen können gerne auch per Email an mich gerichtet werden. Bei Diskussionsbedarf können Sie sich auch in den Übungsstunden an mich wenden.
- Rückkopplung ist ausdrücklich erwünscht !!!!!

1 - Organisatorisches

Fragen?

2 - Einführung

2 - Einführung

Kapitel 2 - Einführung

- Am Anfang war...
- Was ist objektorientierte Programmierung
- Literatur

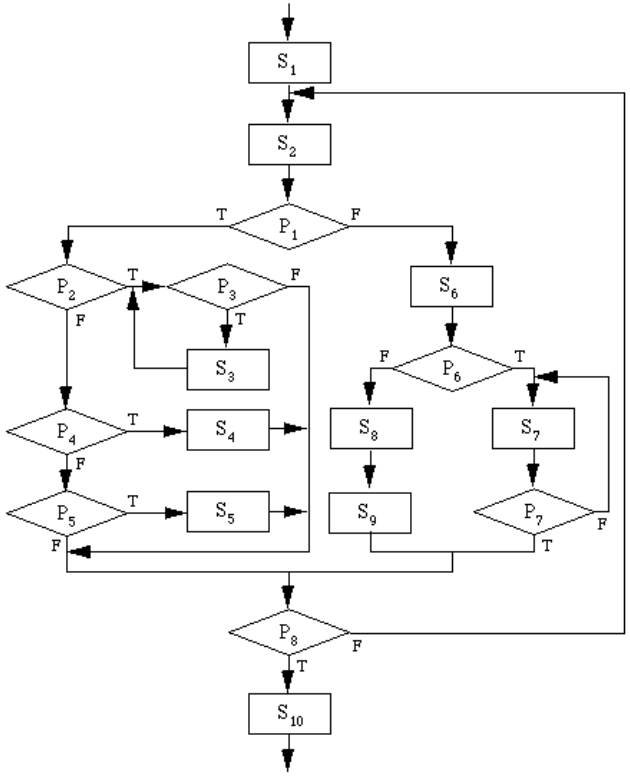
2 - Einführung

Am Anfang war…

…das Programmieren mit Steckverbindungen, Schaltern und Lochkarten

2 - Einführung

Dann kam die Spagetticode Programmierung…

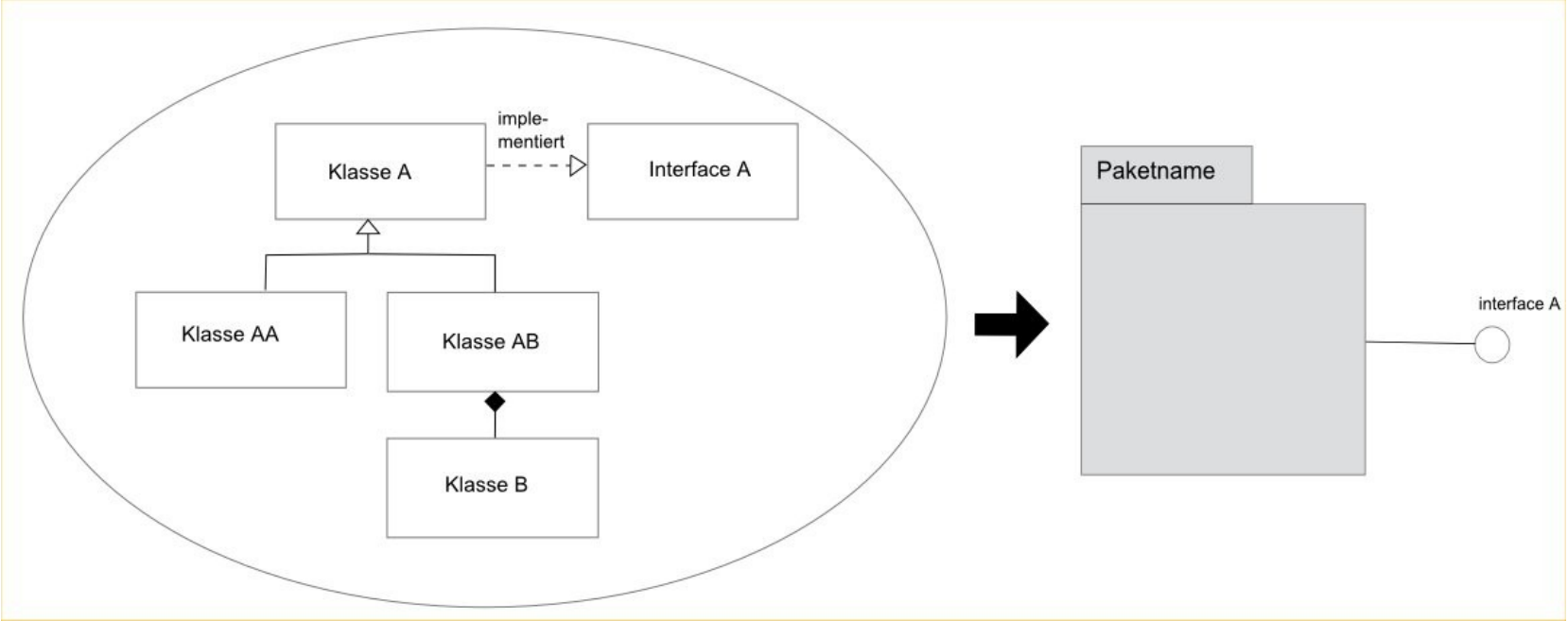


2 - Einführung

Der Bedarf Programmteile wiederverwerten zu können führte zur prozeduralen Programmierung…

2 - Einführung

…die mittlerweile insbesondere bei größeren Projekten, bei denen mehrere Entwickler beteiligt sind, von der objektorientierten Programmierung abgelöst wurde …



2 - Einführung

Was ist objektorientierte Programmierung?

- Objektorientierte Programmierung ist ein Mittel Programme in gekapselter Modulbauweise zu erstellen!
- Dabei entspricht jede Klasse einem Modul das Variablen (Attribute) und Funktionen (Methoden) enthält.
- Mittels Vererbung kann Code effizienter wiederverwendet werden.

2 - Einführung

Literatur

- Bücher zum Thema C bzw C++-Programmierung gibt es unzählige, einige davon sogar online zugängig.
- C++ Bjarne Stroustrup, Addison-Wesley Verlag, ISBN 3-8272-1660-X
- Ulrich Breymann: Der C++-Programmierer; Carl Hanser Verlag, 2. Auflage, 2011
- Objektorientiertes programmieren in C++, Addison-Wesley, ISBN 3-8273-1771-1
- Dietrich May: Grundkurs Software-Entwicklung mit C++;Springer, 2. Auflage, 2006
- de.wikibooks.org/wiki/C++-Programmierung
- www.cplusplus.com
- www.cppreference.com
- www.cpp-tutor.de

3 - Unterschiede C und C++

3 - Unterschiede C und C++

Kapitel 3 – Unterschiede C und C++

- Historie
- Kompatibilität
- Was man in C++ anders macht

3 - Unterschiede C und C++

Historie

- In den Anfängen der Programmierung waren die Programme, die erstellt wurden meist klein (wenige Kilobyte) und in Assembler geschrieben.
- Mit C konnte man dann prozedural und sogar strukturiert programmieren.
- Programme und Bibliotheken konnten von verschiedenen Programmierern verstanden und genutzt werden.
- Erst die Objektorientierte Programmierung z.B. mit C++ ermöglichte ein sicheres reibungsloses Arbeiten an grossen Projekten...

3 - Unterschiede C und C++

Kompatibilität

- Ein C++ Compiler erzeugte früher im ersten Schritt C Code, der erst in weiteren Schritten in Assembler umgewandelt wird.
- So ergibt sich automatisch eine gewisse Kompatibilität von C++ zu C.
- C++ kann sowohl prozedural als auch objektorientiert programmiert werden.
- Die Datenkapselung ist wesentlich konsequenter handhabbar als in C.

3 - Unterschiede C und C++

Was man in C++ anders macht...

In C++ sollten folgende Dinge vermieden werden, da vor allem die Typsicherheit nicht gewährleistet ist:

- #define Textersetzung ohne Typenprüfung (???)
- malloc liefert void pointer zurück, der typenlos ist
- Unions (???)
- void * ist typenlos
- casts Typekonvertierung kann Fehler provozieren (???)

3 - Unterschiede C und C++

Was man in C++ anders macht...

Statt Konstanten mit #defines zu definieren können wir mit const eine Konstante definieren. Speichereffizienter ist jedoch die enum Anweisung:

```
#include <stdio.h>

int main(int argc, char **argv){
  enum Color { red, green, blue };
  Color r = red; // Value range check by compiler
  switch(r)
  {
    case red  : printf("red\n");   break;
    case green: printf("green\n"); break;
    case blue : printf("blue\n");  break;
  }
}
```

3 - Unterschiede C und C++

Was man in C++ anders macht...

Die const Anweisung kann auch in Zusammenhang mit Pointern verwendet werden.

```
char c1 = 'A', c2='B';
char* p1
const char* p2;                 // pointer is protected
char* const p3=&test;           // contents is protected
const char* const p4= &test;    // both protected

p1 = &c1;
*p1 = 'X';
p2 = &c1;     // not possible! Pointer would be changed
*p2 = 'X';
p3 = &c1;
*p3 = 'X';    // not possible! Contents would be changed
p4 = &c1;     // not possible! Pointer would be changed
*p4 = 'X';    // not possible! Contents would be changed
```

3 - Unterschiede C und C++

Was man in C++ anders macht...

In C++ kann auch eine Methode (Funktion) als const definiert werden. Dann können die Klassenattribute nicht durch diese Funktion verändert werden. Beispiel dafür sind getter Methoden.

```
int fkt() const; // class-attributes can not be changed!
```

4 - Neue Sprachmittel

Kapitel 4 – Neue Sprachmittel

- Deklaration für einen Block
- Struct
- Typenkonversion
- Defaultparameter
- Referenzen
- New und delete
- Namespaces

4 - Neue Sprachmittel

Deklaration für einen Block

Zunächst kann man in C++ eine Variable auch inmitten einer Funktion in einem Block deklarieren z.B.:

```
for (int i=0; i<10; i++){  // variable i is only valid insode of
                           // the „for“ loop
}
// or it might be declared inside of a Block
{
 int zahl=0;
 zahl+=1;
} // after the block the variable does not exist anymore!
```

4 - Neue Sprachmittel

Struct

Der Ausdruck struct kann bei der Instanziierung (Deklaration) weggelassen werden

```
struct Schwein{
    int groesse;
    int gewicht;
};
Schwein berta; // in C++ style ...
               // in plain C a struct would be necessary before the
               // Schwein
```

4 - Neue Sprachmittel

Typenkonversion

Es gibt einen Datentyp bool. Die Typenkonversion zwischen bool und int ist wie folgt definiert: Alle Werte ungleich null werden zu true convertiert, alle Werte gleich null werden false.

```
bool b = 100;                  // äquivalent zu b = true;

int ergebnis=false;            // false->0, true->1

bool bIsOk=(b != ergebnis); // ergibt in diesem Fall true
```

4 - Neue Sprachmittel

Default Parameter

Man kann bei Funktionen "Default Parameter" angeben. Sind "Default Parameter" angegeben, so kann man die letzten Argumente beim Aufruf der Funktion weglassen. Diese werden dann mit den Defaultwerten belegt:

```
// Funktionsdeklaration

int function(int a, int b=1, int c=2){

}

function(1,2); // entspricht a=1,b=2,c=2
```

4 - Neue Sprachmittel

Referenzen

Mit vorangestelltem & wird die Deklaration einer Variable zur Deklaration einer Referenz:

```
int Zahl=5;
int &verweisAufZahl = Zahl; // Deklaration einer Referenz!
int pointer*;

verweisAufZahl = 6; // damit ändert man die Variable Zahl auf 6
pointer=&verweisAufZahl; // nun zeigt auch die Variable pointer
                         // auf die Adresse der Variablen "Zahl"
```

4 - Neue Sprachmittel

Besonders häufig werden Referenzen bei Funktionsübergaben verwendet:

```
void addiereUndSubtrahiere(int& erg1, int& erg2,int zahl1, int zahl2)
{
    erg1=zahl1+zahl2;
    erg2=zahl1-zahl2;
}
int main(int argc, char** argv)
{
  int a=1,b=1;
  int erg1,erg2;
  addiereUndSubtrahiere(erg1,erg2,a,b); // equivalent: erg1=a+b;
                                        //             erg2=a-b;
}
```

4 - Neue Sprachmittel

new und delete

Statt wie in ANSI-C mit malloc Speicher zu reservieren bietet C++ die Operatoren "new" und "delete" bzw. "new[]" und "delete[]".

```
int *feld;

feld = new int[100]; // Speicher reservieren

feld[0] = 123;       // Speicher nutzen

delete[] feld;       // Speicher freigeben
```

4 - Neue Sprachmittel

namespaces

- Wenn in einem grossen Projekt z.B. in zwei Packages jeweils eine Funktion mit gleichem Namen und gleichen Übergabeparametern existieren, so kann ein C wie ein C++ Compiler beim Aufruf dieser Funktion nicht entscheiden, welche der zwei Implementationen verwendet werden soll.
- Um existierenden Code so abzukapseln, dass später doch beide Funktionen aufgerufen werden können, benutzt man Namespaces.
- Der Aufruf wird dann mit dem scope operator :: realisiert.
- Anonyme namespaces sind nur im Modul sichtbar: namespace{...}

4 - Neue Sprachmittel

```
namespace package1
{
 void f(){
   // implemetation von package1
 }
}
namespace package2
{
 void f(){
   // implemetation von package2
 }
}
int main(void)
{
 package1::f(); // ruft f von package1 auf
 package2::f(); // ruft f von package2 auf
}
```

5 - Klassen, Objekte und Methoden

5 - Klassen, Objekte und Methoden

Kapitel 5 – Klassen Objekte und Methoden

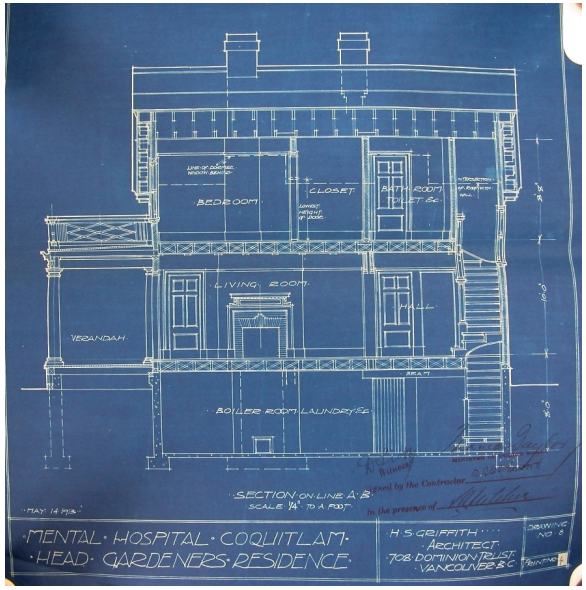
- Was sind Objekte, Attribute und Methoden
- Unterschiede zwischen Class und Struct
- Zugriffsschutz
- Explizit und implizit inline
- Statische Attribute und statische Methoden
- Konstruktoren und Destruktoren
- Der Kopierkonstruktor (Teil 1)
- Weiterleiten von Initialisierungen

5 - Klassen, Objekte und Methoden

Klassen und Objekte

- Eine Klasse ist ein Bauplan während ein Objekt die mit Hilfe des Bauplans erstellte Umsetzung ist.

Klasse



Objekt

5 - Klassen, Objekte und Methoden

Was sind Objekte, Attribute und Methoden

- In der objektorientierten Programmierung werden reale Objekte (z.B. ein Schwein) in einer Klasse modelliert um dann als modelliertes Objekt (z.B."Berta") instantiiert und genutzt zu werden.
- Attribute des Objektes sind Variablen, die das Objekt beschreiben.
- Methoden des Objektes sind Funktionen, die zum Objekt gehören.

5 - Klassen, Objekte und Methoden

Unterschied zwischen Class und Struct

- In C++ kann "struct" durch "class" ersetzt werden.
- Der wesentliche Unterschied zum struct ist, dass innerhalb einer Klasse normalerweise auch die Funktionen implementiert werden, die das Objekt modellieren und auf die Membervariablen zugreifen.
- Ausserdem sind Variablen, die in einer Klasse implementiert werden nur von deren Funktionen zugreifbar, es sei denn man schreibt das Schlüsselwort `public:` davor.
- Membervariablen sind die Variablen, die innerhalb der Klasse deklariert werden und das Objekt beschreiben.

5 - Klassen, Objekte und Methoden

Zugriffsschutz

- Streng genommen greift man in C++ von ausserhalb einer Klasse nur mit Hilfe sogenannter setter und getter Funktionen auf eine Membervariable der Klasse zu.

  Beispiel:

  Variable: gewicht

  Setterfunktion: void setGewicht(double g){…}

  Getterfunktion: double getGewicht(){...}

- Damit ist die Klasse optimal gekapselt.
- Schreibt man private: vor Membervariablen oder Memberfunktionen, so ist der Zugriff von ausserhalb des instantiierten Objektes nicht möglich.
- Die Variablen und Funktionen sind geschützt. Schreibt man public: vor die Variablen oder Funktionen, so kann auch von auserhalb auf die Funktionen und Variablen zugegriffen werden.

5 - Klassen, Objekte und Methoden

Explizit und implizit inline

- Inline Methoden können vom compiler wie ein Makro an die Stelle des Aufrufs kopiert werden.
- Das spart den Overhead beim Aufruf, braucht aber mehr Programmspeicher.
- Wenn die Methoden innerhalb der Klassendeklaration implementiert sind, sind sie implizit inline.
- Mit dem Schlüsselwort **inline** vor einer Memberfunktion kann eine Funktion auch explizit als inline deklariert werden.

5 - Klassen, Objekte und Methoden

```
// Header-File: Schwein.h
class Schwein
{
 private: // kann von extern nur über setter und getter Funktionen zugegriffen werden
  int m_groesse; // Variablen oder Attribute
  int m_gewicht; // Variablen oder Attribute

 public: // mit public können die Funktionen auch von ausserhalb genutzt werden

  Schwein(int groesse=0,int gewicht=0){ // Konstruktor wird beim instantiieren eines Objekts
                                        // aufgerufen
     m_groesse = groesse; // Variablen initialisieren
     m_gewicht = gewicht; // Variablen initialisieren
  }
  ~Schwein(); // Destruktor wird aufgerufen, wenn die Instanz des objektes zerstört wird
  void setGroesse(int groesse);  // Deklaration der Methode,Memberfunktion oder Elementfunktion
  void setGewicht(int gewicht);  // Deklaration der Methode,Memberfunktion oder Elementfunktion
  int getGroesse();             // Deklaration der Methode,Memberfunktion oder Elementfunktion
  int getGewicht();             // Deklaration der Methode,Memberfunktion oder Elementfunktion
}
```

5 - Klassen, Objekte und Methoden

```
// CPP-File: Schwein.cpp
#include "Schwein.h"
Schwein::~Schwein(){ // Destruktor wird aufgerufen, wenn die Instanz des Objektes zerstört wird
// typischerweise stehen hier die delete Aufrufe
}
void Schwein::setGroesse(int groesse){  // Methode,Memberfunktion oder Elementfunktion
  m_groesse = groesse;
}
void Schwein::setGewicht(int gewicht){  // Methode,Memberfunktion oder Elementfunktion
  m_gewicht = gewicht;
}
int Schwein::getGroesse(){              // Methode,Memberfunktion oder Elementfunktion
  return m_groesse;
}
int Schwein::getGewicht(){              // Methode,Memberfunktion oder Elementfunktion
  return m_gewicht;
}
```

5 - Klassen, Objekte und Methoden

```
// --- Main.cpp File ------------------------------------------------
#include "Schwein.h"

int main(void)
{
 Schwein Berta; // Berta wird instantiiert und ist dann ein Objekt oder eine Instanz
                // der Klasse Schwein

 Berta.setGewicht(200);// Berta ist 200kg schwer
 int gewicht=Berta.getGewicht();
}
```

5 - Klassen, Objekte und Methoden

Statische Attribute und statische Methoden

- Methode, die nicht als static deklariert werden bekommen vom Compiler als erstes Argument den **this** pointer, einen Zeiger auf die eigene Instanz der Klasse.
- Statische Methoden bekommen vom compiler keinen **this** pointer.
- Sie können nur auf statische Attribute zugreifen!
- Statische Attribute (Variablen) in einer Klasse existieren genau einmal pro Klasse schon bevor überhaupt eine Instanz der Klasse existiert.

Beispiel: Keine Instanz von einem Konto, aber Zinssatz kann schon festgelegt werden, da setZinssatz() und zinssatz als static deklariert sind! Statische Attribute müssen explizit initialisiert werden.

5 - Klassen, Objekte und Methoden

```
class Konto
{
 float kontostand;
 static float zinssatz;// für alle Instanzen ist der Zinssatz gleich
 public:
 static void setZinssatz(float zinssatz){
    this->zinssatz=zinssatz;
 }
 static float getZinssatz(){
    return zinssatz;
 }
}
float Konto::Zinssatz = 0.0f;
int main(void){
 Konto::setZinssatz(0.3);
 printf("Zinssatz = %f\n",Konto::getZinssatz());
}
```

5 - Klassen, Objekte und Methoden

Konstruktoren und Destruktoren

- Der Konstruktor ist eine Methode, die aufgerufen wird, wenn eine Instanz eines Objekts instantiiert (bzw. deklariert) wird.
- Dort können Membervariablen, also dem Objekt zugehörige Variablen initialisiert werden. Wird das Objekt zerstört, z.B. durch beenden des Programms, so wird der Destruktor aufgerufen.
- Im Destruktor sollten alle mit "new" reservierten Speicher mit "delete" wieder freigegeben werden.

5 - Klassen, Objekte und Methoden

```
// cpp-File
// Konstruktor wird beim instantiieren eines Objekts aufgerufen
Schwein::Schwein(int groesse=0,int gewicht=0){
  // int *feld; // wurde bereits in der Klasse deklariert
  feld = new int[100];// Speicher reservieren
  feld[0]=123;        // Speicher initialisieren
  ...
}
// Destruktor wird aufgerufen, wenn ein Objekt zerstört wird
Schwein::~Schwein(){
  delete[] feld;    // Speicher freigeben
}
```

- Sowohl ein Konstruktor als auch ein Destruktor hat keinen Rückgabewert!

5 - Klassen, Objekte und Methoden

Der Kopierkonstruktor (Teil 1)

- Eine Besonderheit stellt der Kopierkonstruktor dar. Es gibt ihn ohne dass man ihn implementiert:

```
Schwein Berta(200,100);
Schwein Olaf(Berta);  // hier kopiert der Copy-Konstruktor
                      // die Daten von Berta in die Instanz Olaf
Schwein Rudolf=Berta; // hier kopiert der Copy-Konstruktor
                      //die Daten von Berta in die Instanz Rudolf

Rudolf=Olaf; // kopiert alle Variablen von Olaf in Rudolf, aber
             // mittels des = Operators!

fuettere(Olaf); // Call by value: Kopierkonstruktor wird benutzt
Rudolf=groesstesSchwein(); // Return value: Kopierkonstruktor benutzt
```

5 - Klassen, Objekte und Methoden

- Der Kopierkonstruktor kann zu unerwarteten Ergebnissen führen, vor allem wenn im eigentlichen Konstruktor mehr gemacht wird, als die Variablen zu initialisieren.
- z.B. werden Pointer auch nur kopiert und zeigen dann auf die gleichen Speicherbereiche wie die Original Instanz
- Man kann sich z.B. einen eigenen Kopierkonstruktor schreiben (siehe Teil 2):

```
Schwein::Schwein(const Schwein& T) {
  // here the copy-constructor is implemented!!!
}
```

- Möchte man verhindern, dass ein Kopierkonstruktor benutzt wird, so definiert man sich eine Kopierkonstruktor Funktion als private: …

5 - Klassen, Objekte und Methoden

Weiterleiten von Initialisierungen

- Werden Objekte (wie im Unteren Beispiel departure und arrival) innerhalb einer Klasse instantiiert und deren Konstruktor aufgerufen, so kann vom Klasseneigenen Konstruktor Daten über die "Member Initialization List" (mit Komma getrennte Liste hinter Konstruktor) weitergeleitet werden

5 - Klassen, Objekte und Methoden

```
class CflightConnection{
 public:
 //
            Daten werden an weitere Konstruktoren weitergeleitet
 CflightConnection(const char*depart, const char* arrive,
              int flighnumber):departure(depart), arrival(arrive)
 {
 }
 private:
 Ctime departure; // hier wird der Konstruktor mit den Daten
                  // der Weiterleitung des obigen eigenen
                  // Konstruktors gefüttert.
 Ctime arrival;   // hier wird der Konstruktor mit den Daten
                  // der Weiterleitung des obigen eigenen
                  // Konstruktors gefüttert.
}
```

5 - Klassen, Objekte und Methoden

- Dabei könnten „departure“ und „arrival“ auch const oder eine Referenz & sein:

```
const Ctime departure; // const variablen können nur mit Member
                       // Initialization List oder direkt
                       // initialisiert werden
Ctime& arrival;        // referenz kann nur mit Member
                       // Initialization List oder direkt
                       // initialisiert werden
```

6 - Überladen von Funktionen und Operatoren

Kapitel 6 – Überladen von Funktionen und Operatoren

- Überladen von Funktionen
- Der Kopierkonstruktor (Teil 2)
- Name Mangeling
- Extern “c“
- Überladen von Operatoren

6 - Überladen von Funktionen und Operatoren

Überladen von Funktionen

- In C war es nicht möglich Funktionen mit gleichem Funktionsnamen aber unterschiedlichen Parameterzahlen oder Datentypen zu definieren.
- In C++ ist dies möglich:

```
int addiere(int a,int b,int c);   // 1. Funktion
int addiere(float a, float b);    // 2. Funktion
int addiere(char a, double b);    // 3. Funktion
```

- Für den Aufruf entscheidet dabei der Compiler welche der deklarierten Funktionen aufgerufen wird. Z.B. ruft addiere(1.2f,1.3f) die 2.Funktion auf.

6 - Überladen von Funktionen und Operatoren

Überladen von Funktionen

- So kann auch der Konstruktor und der Kopierkonstruktor überladen werden.

```
class MyClass
{
 public:
 MyClass(); // constructor
 MyClass(int a); // overloaded constructor
}
```

6 - Überladen von Funktionen und Operatoren

Der Kopierkonstruktor (Teil 2)

- Der Kopierkonstruktor wird aufgerufen, sobald eine Instanz einer Klasse einer anderen Instanz der selben Klasse zugewiesen wird.
- Das kann entweder durch den = Operator geschehen oder bei der Instanziierung der Klasse:

```
Schein berta;

Schein heinrich(berta); // Kopiert berta nach heinrich

Schwein heinrich = berta; // nutzt auch den Kopierkonstruktor
```

Oder

```
heinrich = berta; // Kopiert berta nach heinrich mittels
                  // der Operator=() Funktion
```

6 - Überladen von Funktionen und Operatoren

- Wurde kein Kopierkonstruktor implementiert, so wird der Standard Kopierkonstruktor verwendet. Dieser kopiert einfach die Inhalte aller Variablen der Quelle in die entsprechenden Variablen des Zielobjektes.
- Mit Vorsicht zu genießen ist die Kopie von Zeigern.
- Im obigen Beispiel wird Schwein `heinrich` als eine Kopie von `Berta` erzeugt. Falls `Berta` Zeiger auf eigene Variablen beinhaltet, so werden diese Zeiger auch kopiert.
- Sie zeigen dann in der Kopie (in diesem Fall Zeiger von `Heinrich`) immer noch auf Variablen von `Berta`.

6 - Überladen von Funktionen und Operatoren

- Eine Implementation eines Kopierkonstruktors kann beispielsweise folgendermassen aussehen:

```
class Date
{
 public:

 int tag,monat,jahr, *ptrToTag;

 Date();                  // Konstruktor
 ~Date();                 // Destruktor
 Date(const Date& d){     // Kopierkonstruktor
    tag   = d.tag;
    monat = d.monat;
    jahr  = d.jahr;
    ptrToTag = &tag;
 }
}
```

6 - Überladen von Funktionen und Operatoren

NameMangeling

- Der Compiler führt ein NameMangling durch, d.h. die Funktion wird mit den aufgerufenen Datentypen in einem String codiert...z.B. addiere@@@int@@@int.
- Da die Syntax des NameMangling von Compiler zu Compiler unterschiedlich ist, kann das zu Problemen bei Bibliotheken führen.
- Nutzt man `extern"c"` so können Funktionen nicht überladen werden, aber ein NameMangling findet auch nicht statt.

6 - Überladen von Funktionen und Operatoren

extern ”c“

```
extern "c"
{
 int addiere(int a, int b);
}
```

- So ist es möglich auch C oder C++ Funktionen zu nutzen, die mit einem anderen Compiler geschrieben wurden.

6 - Überladen von Funktionen und Operatoren

Überladen von Operatoren

- Nicht nur Funktionen können überladen werden, sondern auch Operatoren.
- Die Syntax einer Funktion, die einen Operator überlädt lautet wie folgt:

  wenn der Operand links vom Operator nicht einer selbst geschriebenen Klasse angehört:

```
Rückgabetyp operator+ (datentyp leftHandSide, datentyp rightHandSide)
```

wenn der Operand links vom Operator einer selbstgeschriebenen Klasse angehört und in der Klasse überladen wird:

```
Rückgabetyp operator+ (datentyp rightHandSide)
```

6 - Überladen von Funktionen und Operatoren

Überladen von Operatoren: Bedingungen

- Nicht überladen werden können:

```
?:      Bedingter Ausdruck
sizeof  Sizeof Operator
.       Punktoperator
::      Scopeoperator
.*      Komponentenzeiger
```

6 - Überladen von Funktionen und Operatoren

```
class vector{
  const int SIZE=3;
  double v[SIZE];
public:
  vector(double x,double y,double z)
  {
  v[0]=x;   v[1]=y;    v[2]=z;
  }
  // --------------- Operator überladen ------------ //
  vector operator+(const vector& v2){
    vector t(0,0,0);
    for (int i=0;i<SIZE;i++){
       t.v[i]=v[i]+v2.v[i];
    }
    return t;
  }
};
// ---------------------- main ---------------------//
int main(void) {
 vector v1(1,2,3), v2(4,6,5), v3(0,0,0);
 v3 = v1+v2; // funktioniert nur weil der Operator+ überladen ist
}
```

6 - Überladen von Funktionen und Operatoren

Überladen von Operatoren: Bedingungen

- Mindestens ein Operand muss abstrakter (z.B. selbst definierter) Datentyp sein.
- Die Regeln der C Operatoren sollten eingehalten werden (z.B.Auswertereihenfolge).
- Die Anzahl der Parameter ist vorgegeben.
- Überlädt man beipielsweise den = Operator, so sollte man darauf achten, dass die operator= Methode auch einen Rückgabewert hat (in diesem Fall *this), denn sonst ist obj1=obj2 möglich, nicht aber obj1=obj2=obj3.

6 - Überladen von Funktionen und Operatoren

Überladen von Operatoren: Bedingungen

- Für den Fall, dass der lefthand Operator nicht vom Typ der Klasse ist, gibt es keine Möglichkeit innerhalb der Klasse eine Operatorüberladung zu implementieren.
- Für einen solchen Fall kann man die Operatorüberladung auch ausserhalb der Klasse schreiben.
- Die Klasse, von dessen Typ der erste operator ist, wird dann normalerweise als friend deklariert, und kann somit auch auf private: Elemente zugreifen.
- Auch hier gilt, dass man für call by value und für den Rückgabewert einen Kopierkonstruktor braucht.

3 – Die Programmiersprache C/C++

Beispiel: Skalar mal Vektor, Operatorüberladung ausserhalb der Klasse

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

class Vektor{
public:
 double v[3];
 Vektor(double a, double b, double c){
   v[0]=a;
   v[1]=b;
   v[2]=c;
 }
 void print(){
   printf("Vector = (%f,%f,%f)\n",v[0],v[1],v[2]);
 }
};
```

3 – Die Programmiersprache C/C++

Beispiel: Skalar mal Vektor, Operatorüberladung ausserhalb der Klasse

```
Vektor* operator*(int i, Vektor v2){
  Vektor *result = new Vektor(0,0,0);
  result->v[0]=i*v2.v[0];
  result->v[1]=i*v2.v[1];
  result->v[2]=i*v2.v[2];
  return result;
}

int main(int argc, char* argv[]){
  Vektor* res;
  Vektor a(1,2,3);
  res = 5*a;
  res->print();
  return 0;
}
```

6 - Überladen von Funktionen und Operatoren

Man kann z.B. auch einen Typecast überladen/ einbauen:

```
class Bruch{
public:
 int m_zaehler,m_nenner;
 Bruch(int zaehler, int nenner) : m_zaehler(zaehler),
                                  m_nenner(nenner)
 {}// Konstruktor mit initialisierung von m_nenner und m_zaehler

 operator float(){   // typecast implementieren!!!
   return ((float)(m_zaehler)/(float)(m_nenner));
 }
 operator double(){   // typecast implementieren!!!
   return ((double)(m_zaehler)/(double)(m_nenner));
 }
}
```

6 - Überladen von Funktionen und Operatoren

- Zusätzlich könnten die Operatoren überladen werden, die das Bruchrechnen übernehmen:

```
Bruch operator*(Bruch bruch)
 {
   Bruch ergebnis;
   ergebnis.m_zaehler=this->m_zaehler*bruch.m_zaehler;
   ergebnis.m_nenner=this->m_nenner*bruch.m_nenner;
   return ergebnis;
 }
 Bruch operator+(Bruch bruch)
 {
   Bruch ergebnis;
   ergebnis.m_zaehler=this->m_zaehler*bruch.m_nenner +
                      bruch.m_zaehler*this->m_nenner;
   ergebnis.m_nenner=this->m_nenner*bruch.m_nenner;
   return ergebnis;
 }
```

7 - Vererbung

7 - Vererbung

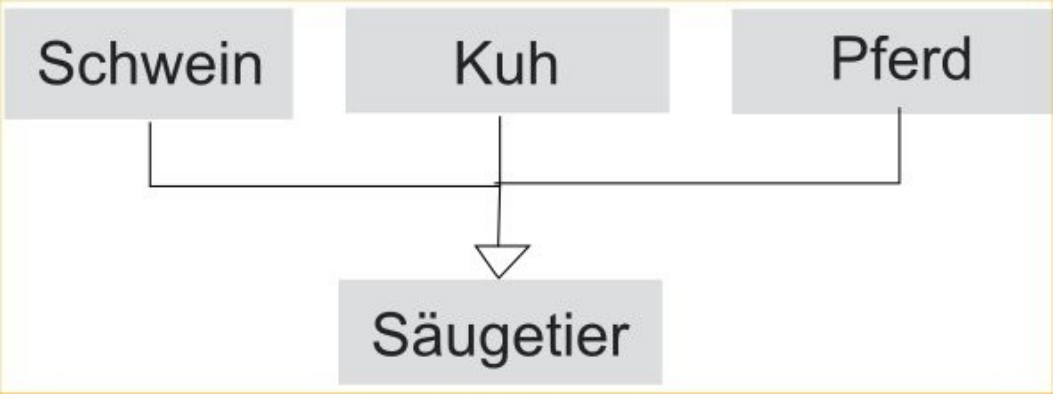
Kapitel 7 – Vererbung

- Grundlagen
- Überschreibung von Funktionen
- Aufruf der Konstruktoren
- Friend
- Virtuelle Funktionen und Polymorphismus
- Zugriffsrechte
- Nachtrag Kopierkonstruktor (Teil 3)

7 - Vererbung

Grundlagen

- Besonders effizient programmieren kann man mit Hilfe von Abstraktion. Wenn zum Beispiel "Kuh", "Schwein" und "Pferd" = "Saeugetiere" sind könnte man zunächst die Klasse Säugetiere implementieren und davon "Kuh", "Schwein" und "Pferd" ableiten.



7 - Vererbung

- In anderen Worten: Man kann die Attribute und Methoden, die für alle Säugetiere gelten an die einzelnen Saeugetiere vererben.
- Einzelne Methoden, wie z.B. das "grunzen()" des Schweines, könnten dann für das vom Säugetier abgeleitete Schein extra implementiert werden
- Konstruktoren und Destruktoren werden nicht vererbt

7 - Vererbung

```
class Saeugetier // Basisklasse
{
 friend Class CLieberGott; // Freunde können auf alle Attribute
         // der Basisklasse zugreifen, auch auf die privaten!
 public:
 void fressen();
 void wachsen();
 protected:      // auf protected: Attribute kann von der
                 // abgeleiteten Klasse zugegriffen werden
 int groesse;
 int gewicht;
 private:        // auf private: Attribute kann von der abgeleiteten
                 // Klasse nicht zugegriffen werden
}
class Kuh : public Saeugetier  // von Saeugetier  abgeleitete Klasse
{
 public:
 void muhen();
}
```

7 - Vererbung

Und noch das Schwein und das Pferd...

```
class Schwein : public Saeugetier{    // von Saeugetier
                                      // abgeleitete Klasse
 public:
 void grunzen();
}

class Pferd : public Saeugetier{     // von saeugetier
                                     // abgeleitete Klasse
 public:
 void wiehern();
}
```

7 - Vererbung

Überschreiben von Funtionen

- Zusätzlich kann man in der abgeleiteten Klasse eine Funktion der Basisklasse überschreiben.
- Das heißt man kann eine Funktion mit gleichem Namen und gleichen Parametern implementieren.
- Bei einer Instanz der abgeleiteten Klasse wird dann diese Funktion statt der überschriebenen Basisklassenfunktion aufgerufen.

7 - Vererbung

Aufruf der Konstruktoren

- Bei Instanziierung eines Objektes einer abgeleiteten Klasse wird zuerst der parameterlose Konstruktor der Basisklasse aufgerufen, dann der Konstruktor der abgeleiteten Klasse.
- Bei mehrfach vererbten Klassen wird zuerst der parameterlose Konstruktor der Basisklasse, dann der der abgeleiteten Klasse, dann der der davon abgeleiteten Klasse... bis zum Aufruf des Konstruktors der aufgerufenen Klasse (der auch Parameter beinhalten kann).

7 - Vererbung

Friend

- Im obigen Beispiel wird die Klasse "CLieberGott" als "friend" deklariert.
- Dadurch kann jede Methode der Klasse "CLieberGott" alle Attribute der Klasse "Saeugetier" ändern und lesen, auch wenn sie **private** sind!

  Anmerkung: Mit **friend:** definierte Klassen werden nicht als friend mit vererbt.

7 - Vererbung

Virtuelle Funktionen und Polymorphismus

- Arbeitet man mit Zeigern vom Typ Basisklasse "Saeugetier" auf eines seiner abgeleiteten Klasseninstanzen z.B. "Schwein", so kennt der Compiler nicht die Funktion "grunzen()", da es keine Funktion der Basisklasse ist.
- Wenn man die Methode "grunzen()" als virtuell in der Klasse Saeugetier deklariert und imlementiert, so kennt der Compiler auch diese Funktion. (z.B. virtual void grunzen();)
- Schreibt man in der Basisklasse hinter einer virtual void Funktion()=0, so muss diese Funktion in jeder abgeleiteten Klasse implementiert werden! Man spricht dann von rein virtuellen Funktionen.
- Eine Klasse, die ein oder mehrere rein virtuelle Methoden enthält kann nicht instanziiert werden

7 - Vererbung

- **Anmerkung:** Mit virtuellen Methoden werden Virtuelle Methodentabellen angelegt und für jede Instanz der Klasse Zeiger auf die Virtuelle Methodentabelle. So erkauft man sich die Flexibilität der virtuellen Methoden besonders bei vielen Instanzen mit Speicherplatz und Laufzeit.

- **Anmerkung:** Klassen, die mindestens eine virtuelle Funktion mit "=0" implementieren heißen abstrakte Klassen!

7 - Vererbung

```
class Saeugetier
{

 private:
 int groesse;
 int gewicht;

 public:
 void fressen();
 void wachsen();
 virtual void grunzen();
 virtual void wiehern();
 virtual void muhen();

}
```

7 - Vererbung

Zugriffsrechte

- public: Auf public Member kann von überall drauf zugegriffen werden!
- protected: Auf protected Member kann von der Klasse, in der sie deklariert sind und deren abgeleiteten Klassen zugegriffen werden!
- private: Auf private Member kann nur von der Klasse zugegriffen werden, in der sie deklariert sind!
- Hinter der abgeleiteten Klasse steht mit : getrennt die Art der Ableitung. Die Art kann private, protected oder public sein. Damit kann man die geerbten Funktionen und Variablen noch mal extra schützen. Mit public werden die Zugriffsrechte der Basis beibehalten, mit protected sind sie mindestens protected, mit private sind sie alle private!

7 - Vererbung

Ableitung	Privileg der Basisklasse	Privileg der abgeleiteten Klasse
:public	public:	public
	protected:	protected
	private:	Kein Zugriff
:protected	public:	protected
	protected:	protected
	private:	Kein Zugriff
:private	public:	private
	protected:	private
	private:	Kein Zugriff

7 - Vererbung

Nachtrag Kopierkonstruktor (Teil 3)

- Wenn man bei der Instanziierung einer Basisklasse die von ihr abgeleiteten Klassen Instanz zuweist, so hat der Compiler alle Daten zur Verfügung um die Basisklasse zu füllen. Andersrum würde der Compiler eine Fehlermeldung ausgeben, da bei der abgeleiteten Klasse mehr Attribute sein könnten, die dann nicht aus der Basisklasse kopiert werden können.

```
// so geht's...

CAbgeleiteteKlasse abgeleitet;

CBasisKlasse basis=abgeleitet; // ruft den Kopierkonstruktor und
   kopiert alle Attribute die die Basisklasse benötigt
```

8 - Templates

8 - Templates

Kapitel 8 – Templates

- Grundlagen
- Funktionstemplates
- Klassentemplates

8 - Templates

Grundlagen

- Manchmal wünscht man sich Funktionen, die mit einem beliebigen Datentyp arbeiten. Diese Möglichkeit bieten Templates (Schablonen)
- Es gibt Funktionstemlates und Klassentemplates

8 - Templates

Funktionstemplates

- Hier ein Beispiel einer Funktion, die das Maximum zweier Instanzen zurückgibt. Wird die Templatefunktion mit verschiedenen Datentypen aufgerufen, so erzeugt der Compiler die Funktion für jeden der genutzten Datentypen neu.

```
template <typename T>
T max(T parameter1, T parameter2)
{
 if (parameter1>parameter2){
    return parameter1;
 }
 else{
    return parameter2;
 }
}
```

Werden mehr als ein Templatedatentyp verwendet, so kann man diese mit Komma innerhalb der spitzen Klammern abtrennen.

```
template <typename T, typename U>
```

8 - Templates

Funktionstemplates

- Der Aufruf dieser Funktionstemplates kann so aussehen wie bisher:

```
int a = max(5,7);
```

  In dem Fall wird 5 und 7 als Zahlen des Typs Integer interpretiert.

- Man kann den Datentyp jedoch auch explizit angeben:

```
int a = max <int> (5,7);
```

8 - Templates

Klassentemplates

- Bei Klassentemplates können Attribute der Klasse und Funktionsparameter und Rückgabewerte der Methoden vom Template-Typ sein.
- Templateklassen können vererbt werden und können virtuelle Funktionen enthalten!

8 - Templates

```
template <class T>
class CMathVector
{
public:
  enum{MAX=3}
  T mathVector[MAX];

  T mittelwert(void)
  {
    T sum=0;
    for (int i=0;i<MAX;i++){
        sum+=mathVector[i];
    }
    return sum/MAX;
  }
}
```

```
// im cpp File Instantiierung:

int main(void)
{
CMathvector <double> dblVekt;
double mittel;

dblVekt.mathVector[0] = 1.234;
dblVekt.mathVector[1] = 0.5643;
dblVekt.mathVector[2] = 0.543;
mittel = dblVekt.mittelwert();
return 0;
}
```

8 - Templates

```
// alternative Implementierung
// ausserhalb der Klassendefinition
template <class T>
T CMathVector<T>::mittelwert(void)
{
  T sum;
  for (int i=0;i<MAX;i++){
      sum+=mathVector[i];
  }
  return sum/MAX;
}
```

9 - Streams

9 - Streams

Kapitel 9 – Streams

- Grundlagen
- Datei schreiben
- Datei lesen

9 - Streams

Grundagen

- Bindet man <iostream> ein, so kann man Zeichen von der Tastatur einlesen und auf dem Bildschirm ausgeben. Dazu nutzen wir den << und den >> Operator:

```
#include <iostream>
using namespace std;
int main(void)
{
 int zahl;
 cout << "geben Sie eine Zahl ein:"; // Der Text in Anführungszeichen wird
   auf
                                    // dem Bildschirm ausgegeben
 cin >> zahl;      // hier wird eine Zahl von der Tastatur eingelesen,
                   // die mit der Eingabetaste bestätigt werden muss
 cout << "Sie haben folgende Zahl eingegeben:" << zahl << endl;
                   // endl ist Zeilenumbruch und forciert das ausgeben
                   // bzw. leeren des Ausgabebuffers
 return 0;
}
```

9 - Streams

Datei schreiben

- In C++ gibt es eine Stream Erweiterung, die für das File-Handling zuständig ist:

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
 ofstream outFile; // für Dateien, die geschrieben werden
 outFile.open("test.txt",ios::out); // öffnen der Datei zum schreiben
 if (!outFile){
    cout << "Ausgabedatei kann nicht geöffnet werden!\n\n";
 }
 else{
   outFile << "Dies ist ein BeispielTEXT" << flush;
 }
 outFile.close();
}
```

9 - Streams

Datei lesen

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main(void){
 string inp;
 ifstream inFile;  // für Dateien, die geöffnet werden

 inFile.open("test.txt",ios::in); // öffnen der Datei zum lesen
 if (!inFile){
    cout << "Eingabedatei kann nicht geöffnet werden!\n\n";
 }
 else{
    while(inFile.getline(inp,1)){
       cout << inp << flush; // flush bedeutet Buffer lehren
    }
 }
 inFile.close();
}
```

10 - Strings

10 - Strings

Kapitel 10 – Strings

- Vorteile gegenüber C-Strings
- Beispiel

10 - Strings

Grundlagen

- Der Vorteil von String zum ursprünglichen C-Sting oder array von char ist:
  - \0 kann im string vorkommen ohne das es das Ende des Strings makiert!
  - es gibt zahlreiche Möglichkeiten, mit denen man Strings manipulieren kann z.B. kann man strings addieren
  - Man kann den String mit rezise vergrößern
  - alle Methoden sind sicher!!!
- Um die Stringklasse benutzen zu können muss sie zunächst includiert werden.

10 - Strings

Beispiel

```
#include <string>
int main(void)
{
 string Nachname("Walter");
 string Vorname("Dieter");
 string Name;
 Name=Vorname+" "+Nachname;
 if (Vorname=="Otto"){
    cout << Name << endl;
 }
 if (Vorname.find("ie")>0){
    cout << "Die Buchstaben ie befinden sich an Stelle "
         << Vorname.find("ie") << "im Vornamen" << Vorname
         << "." << endl;
 }
 return 0;
}
```

11 – Casting und Fehlerbehandlung

11 - Casting und Fehlerbehandlung

Kapitel 11 – Casting und Fehlerbehandlung

- Casting
- Fehlerbehandlung

11 - Casting und Fehlerbehandlung

Casting

Es gibt in C++ weitere Cast Operatoren:

```
static_cast<xyz*>(p)              // prüft Verwandschaftsbeziehung

reinterpret_cast<char*>(&org);    // hier kann zu etwas komplett anderem gecastet
                                  // werden.

const_cast<char*>(q)='B';         // selbst konstanten können überschrieben werden!

dynamic_cast<CKlasse*>(pA)        // Prüft zur Laufzeit ob cast korrekt und gibt
                                  // sonst NULL zurück
```

- Beim dynamic_cast wird eine vtable (virtuelle Methodentabelle) in das Programm eingebunden. Damit ist der Dynamic_cast für embedded Systeme ungeeignet.

11 - Casting und Fehlerbehandlung

Fehlerbehandlung

Es gibt zweierlei Arten der Fehlerbehandlung in C++:

1. Mittels assertion

```
#include <assert.h>
...
assert(true);  // führt keine Fehlerbehandlung aus
assert(false); // führt eine Fehlerbehandlung aus
```

11 - Casting und Fehlerbehandlung

2. Mittels Exceptions try-catch:

```
#include <iostream>
int i=0,j=0;
try
{
   i=i/j;
}
catch(...) // statt ... kann hier auch ein Ausnahmetyp stehen
{  // springt mittels einer exception hier rein um den Fehler zu behandeln!
   std::cout << "Division durch Null!"<<flush;
}
```

12 – Der neue C++ Standard

12 - Der neue C++ Standard

Kapitel 12 – Der neue C++ Standard

- Neue Schlüsselworte
- Die range basierte For Schleife
- funktionale Programmierung (Lambda Funktionen)
- generische Programmierung (zusätzliche Möglichkeiten)
- Multithreading
- Atomare Datentypen
- Asynchrone Aufgaben
- Reguläre Ausdrücke

12 - Der neue C++ Standard

Neue Schlüsselworte

Um die Programmierung zu vereinfachen wurden neue Schlüsselworte eingeführt:

Beim neuen Datentyp **auto** wird der eigendliche Datentyp erst bei der Initialisierung der Variablen definiert. Das macht besonders bei Templates Sinn.

```
auto var = x + y; // sind x und y Objekte einer Klasse,
                  // so ist auch var Objekt dieser Klasse
```

Um den Datentyp zu ermitteln liefert decltype den Datentyp zurück.:

```
typedef decltype(var) myDataType;
myDataType var2; // var2 hat den selben Datentyp wie var
```

12 - Der neue C++ Standard

Die range basierte For- Schleife

Um sich wenig um die Größe von Feldern kümmern zu müssen gibt es nun die range basierte for Schleife:

```
std::vector<int> vec({ 1, 2, 3, 4, 5, 6 });
for (const auto x : vec){
     std::cout << x << std::endl;
}
```

Die Schleife iteriert über alle Elemente vec.

12 - Der neue C++ Standard

Funktionale Programmierung

Ab C++11 sind auch die sogenannten Lambda-Expressions der funktionalen Programmierung möglich.

```
std::vector<int> v = {15, -2, 4, -1};
std::sort(v.begin(), v.end(), [](int a, int b) {
   return abs(a)<abs(b); });
```

Innerhalb der Funktionen kann man zwar Membervariablen auslesen und sich Hilfsvariablen deklarieren und benutzen. Es können aber keine Membervariablen beschrieben werden.
In Eckigen Klammern [] wird die Bindung zu den lokalen Variablen angegeben, in runden Klammern () die Parameter

12 - Der neue C++ Standard

Generische Programmierung

Als generischer Programmiererweiterung werden ab C++11 folgende eingeführt, hier aber nicht näher beschrieben:

- Templates, die beliebig viele Parameter annehmen
- Zusicherungen, die zur Compilezeit ausgewertet werden
- Konstanten, die zur Compilezeit ausgewertet werden
- Aliase Templates, um einfache Namen für teilweise gebundene Templates zu definieren

12 - Der neue C++ Standard

Multithreading

Es gibt im Wesendlichen 3 Varianten einen Thread zu erzeugen:

1. mit Hilfe einer Funktion
2. mit Hilfe eines Objektes
3. mit Hilfe einer Lambda Expression

12 - Der neue C++ Standard

Multithreading (mit Funktion)

```
#include <iostream>
#include <thread>
void helloFunction() {
  std::cout << "C++11 Thread with function"<< std::endl;
}
int main() {
 std::thread t1(helloFunction);
 t1.jion();
}
```

12 - Der neue C++ Standard

Multithreading (mit Objekt)

```
#include <iostream>
#include <thread>
class HelloFunctionObject {
  public:  void operator()() const {
        std::cout << "C++11 Thread with Object" << std::endl;
  }
}
int main() {
 HelloFunctionObject helloFunctionObject;
 std::thread t2(helloFunctionObject);
 t2.jion();
}
```

12 - Der neue C++ Standard

Multithreading (mit Lambda Ausdruck)

```
#include <iostream>

#include <thread>

int main() {

 std::thread t3([]{std::cout << "C++11 Thread with lambda
   function" << std::endl;});

 t3.jion();

}
```

12 - Der neue C++ Standard

Atomare Datentypen

- Atomare Datentypen sind Datentypen, die nicht aus anderen Datentypen zusammengesetzt sind.
- Auf diese Datentypen können atomare Operationen ausgeführt werden.
- Atomare Operationen sind Operationen, die nicht unterbrochen werden können.
- Es gibt bereits viele atomic Datentypen, die einen äquivalenten build in Typ haben: z.B. atomic_bool, atomic_char, atomic_int etc.
- Mit Hilfe des `std::atomic-Klassen-Template` ist es möglich eigene atomare Typen zu implementieren.

12 - Der neue C++ Standard

Asynchrone Aufgaben

- Sie sind unter dem Namen Futures bekannt
- Der Compiler kümmert sich selbst um die Verwaltung und prüft ob es z.B. sinnvoll ist einen neuen Thread zu erzeugen
- std::async liefert ein std::future Objekt zurück, von welchem man zu einem späteren Zeitpunkt das Ergebnis abrufen kann
- Sollte das Ergebnis noch nicht vorliegen blockiert der Thread

12 - Der neue C++ Standard

Asynchrone Aufgaben(Beispiel)

```
#include <future>

int product(int a, int b) {
 return a*b;
}

int main() {
 int a = 20;
 int b = 10;
 std::future<int> futureSum = std::async([=](){return a+b;});
 auto futureProduct = std::async(&product,a, b);
 std::cout << futureSum.get() << std::endl;
 std::cout << futureProduct.get() << std::endl;
}
```

12 - Der neue C++ Standard

Reguläre Ausdrücke

Mit Hilfe der regukären Ausdrücke können Zeichenfolgen durchsucht und ersetzt werden.

Ab C++11 werden reguläre Ausdrücke der folgenden Grammatiken unterstützt:

- ECMAScript
- basic
- extended
- awk
- grep
- egrep