Fachhochschule Köln
Cologne University of Applied Sciences

Angewandte Mathematik und Programmierung

Einführung in das Konzept der objektorientierten Anwendungen zu wissenschaftlichen Rechnens mit C++ und Matlab

SS2013

Fomuso Ekellem

Inhalt

- **Bis jetzt:**
  - Entwicklungsumgebung
  - Datentypen
  - Operatoren
  - Ausdrücke
  - Schleifen
  - Strukturen
  - Zeiger
  - Referenzen
- **Heute:**
  - **Objekt orientierte Programmierung**

OOP

- **Programmierungsparadigmen (Was ist Objekt orientierte Programmierung)**
- **Grundbegriffe (Wie geht man um)**
  - Objekte
  - Klassen
  - Wichtige Begriffe
- **Klassen und Strukturen in C++ (Wo ist in C++ etwas anders)**
- **Konstruktoren (eine spezielle Methode)**
- **Destruktoren**
- **Zugriffsrechte(private, public und protected)**
- **Zugriffrecht(friend)**
- **Methoden**
  - Was sind Methoden(Funktionen)
  - Definition und Deklaration von Methoden(Funktionen)
  - Parameter und Rückgabewerte
  - Lokale, globale, klassen und statische lokale Variablen
  - Rekursive FunktionenInhalt

Programmierungsparadigmen

- Der Schwerpunkt in dieser Vorlesung liegt auf der Vermittlung **objektorientierter Programmiermethoden,** in einfacher und anschaulicher Form, anhand zahlreicher konkreter Beispiele und Übungsaufgaben.
- **Wichtig!** Ob die Programmiersprache nun C++, Java, Visual Basic oder einen anderen Namen trägt, ist sekundär; Wichtig ist das Verständnis der dahinter stehenden **Grundkonzepte**.
- Objektorientierte Programmierung ist ein **Programmierungsparadigma**.
- **Paradigma**: Ein Beispiel, das als Muster oder Modell dient.
- Vier vorwiegend bekannte Haupttypen:
  - Prozedurale/Imperative
  - Logische
  - Funktionale
  - Objektorientierte

Programmierungsparadigmen

- Viele von Ihnen haben bisher nur prozedurale programmiert.
- **Imperative /prozedurale Paradigma:** Die Funktionen stehen im Vordergrund. Mit bedingten Anweisungen und Sprung-Anweisungen können Programmteile übersprungen oder wiederholt werden.
- Bei prozeduralen Programmiersprachen werden ***zu lösende Probleme*** in Teilprobleme aufgeteilt - auch ***Funktionen*** (C/C++) bzw. ***Prozeduren*** (Modula, PASCAL) genannt.

Programmierungsparadigmen

- **Funktionale- Paradigmen:** Menge von Funktionsdefinitionen und einem Ausdruck. Typischer Vertreter ist die Sprache LISP.
- **Logische-Paradigmen:** Hier werden nur Fakten und Regeln angegeben. Problemlösung wird nicht genauer spezifiziert, sondern vom Interpreter-Programm erstellt. Man kann Logische Paradigmen in Prolog und SQL sehen.
- **Unterschiede:**
  - Funktionale und logische Stille trennen sehr klar die WELCHE Aspekte eines Programms (Programmierer Verantwortung) und die WIE Aspekte (Durchführungsbeschlüsse).
  - Imperative/prozedurale und objektorientierte Programme enthalten im Gegensatz sowohl die Spezifikation und die Details der Implementierung, sie sind untrennbar miteinander verbunden.

Paradigmen- Objektorientierung

- Hier stehen die Daten (Eigenschaften) und nicht die Funktionen, Prozeduren oder Methoden des Programms im Vordergrund .
- Die Daten werden in Objekten gekapselt (Information hiding), die **auch** über Funktionalitäten verfügen, um die Daten zu ändern.
- In diesem Zusammenhang spricht man jedoch nicht von **Funktionen**, sondern von **Methoden**.
- Programme werden aus verschiedenen Objekten aufgebaut.
- Beispiel-Sprachen:

  C Sprache wurde im Hinblick auf objektorientierte Programmierung zu C++ weiterentwickelt. C++ ist hybrid aus imperativem/Prozeduralem C und objektorientierten Erweiterungen aufgebaut.
- Heute ist Java neben C++ „state of the Art“.

OOP-Grundkonzept

- Identifikation von Objekten und Implementierung von Objekt-Typen in Klassen.
- Zuweisung von Aufgaben zu diesen Objekten (Methoden).
- Erzeugte Objekte von Klassen, kommunizieren mit anderen Objekten gleichen Typs durch Senden von Nachrichten über Methoden.
- Die Nachrichten werden ebenfalls von den Methoden des Objekt-Typs empfangen und verarbeitet.
- Siehe Beispiel:

Grundbegriffe- Objekte

- Ein "Objekt" ist alles, was diesem Konzept entspricht,

  oder

- ein Objekt repräsentiert ein Individuum, identifizierbaren Artikel, Einheit, oder Rechtsträger, entweder real oder abstrakt, mit einer klar definierten Rolle bei der Problem-Domäne.

  Beispiele:

  - Materielle Dinge - wie ein Auto, Drucker, ...
  - Rollen - als Mitarbeiter, Chef, ...
  - Vorfälle - wie Flug-, Überlauf-, ...
  - Interaktionen - als Vertrag, den Verkauf, ...
  - Technische Daten - wie Farbe, Form, ...

Grundbegriffe- Objekte

- Die zwei Teile eines Objekts
  - Objekt = Daten + Methoden

- In anderen Worten: Ein Objekt hat die Verantwortung, zu wissen und die Verantwortung zu tun. Im Gegensatz dazu obliegen in der Sprache C diese Aufgaben dem Programmierer.

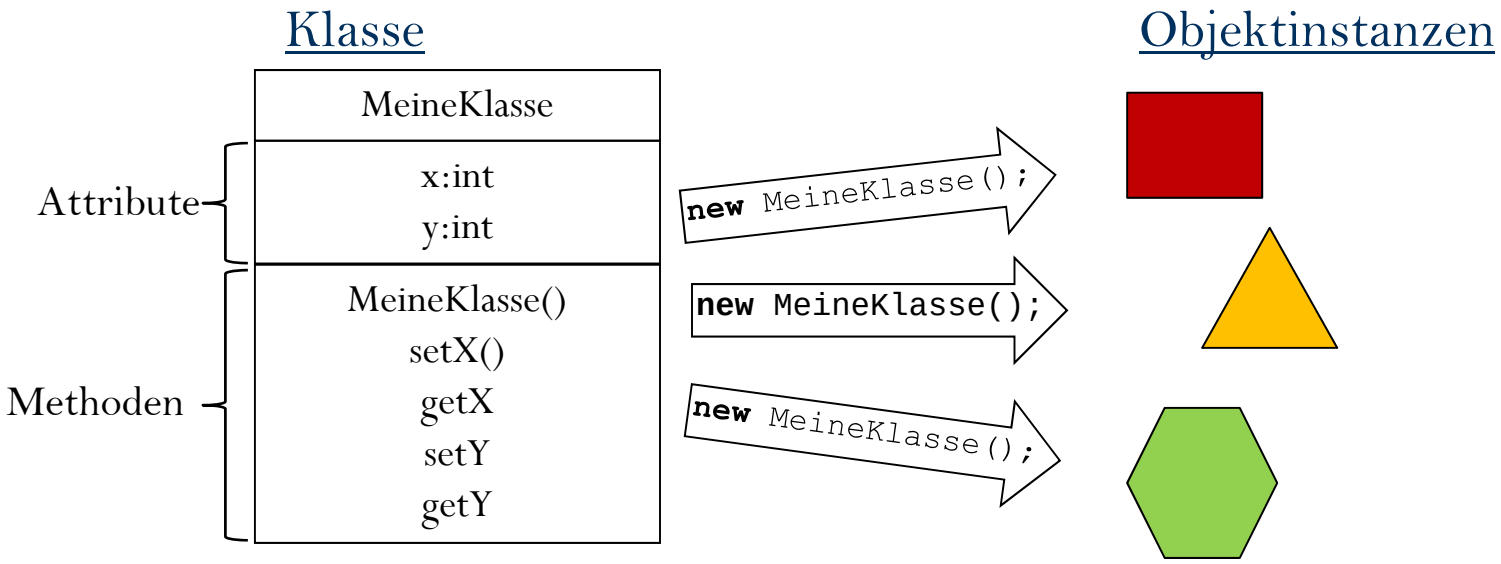
Grundbegriffe - Objekte

Warum Objekte?

- **Modularisierung** - große Software-Projekte lassen sich in kleinere Stücke teilen.
- **Wiederverwendbarkeit** - Programme können aus bereits geschrieben Softwarekomponenten zusammengesetzt werden.
- **Erweiterbarkeit** - Neue Software-Komponenten können geschrieben oder aus bestehenden weiterentwickelt werden.
- Objektdaten **Abstraktion** bezeichnet den Prozess, eine Darstellung wesentlichen Attribute oder Merkmale herauszufinden und die unwesentlichen wegzulassen.
- **Datenkapselung**: Daten (Attribute) sind nicht sichtbar aber Methoden zur Manipulation der Attribute sind sichtbar. So kann nichts „Unsinniges“ mit den Daten passieren.
- OOP erfüllt mit der Unterstützung von **Modularisierung**, **Wiederverwendbarkeit**, **Erweiterbarkeit**, **Abstraktion** und **Kapselung** von Objekten die zentralen Anforderungen, um die immer komplexere Anwendungslandschaft beherrschbar zu machen.

Grundbegriffe-Klassen

Objekte basieren auf Klassen, die einen Bauplan für ein Objekt und dessen Attribute(Daten) und Methoden festlegen.



Klassen fassen gleichartige Objekte zusammen.

Grundbegriffe – Objekte/Klassen

Zusammenfassung:

- Objekte = Daten + Methoden
- Daten (Attribute) = Eigenschaften
- Methode = Operationen rund um das Objekt
- Klassen (Objektkapsel) = Program-Bauplan (Implementierte Objekt in einer Programmiersprache)

Wichtige Begriffe

- **Vererbung**
- **Polymorphie**
- **Virtuelle Klassen und Methoden**
- **Abstrakte Klasse**
- **Templates**
- **...**

Klassen und Strukturen in C++

- Strukturen werden auch in C++ weiterhin als öffentlichen Gruppierung von verwandte Daten benutzt, damit die Umsetzung von C-Quellcode nach C++ auch vereinfacht wird. Da die Strukturen die wichtige Kapselung nicht unterstützen, sollten Sie stattdessen immer Klassen einsetzen.
- In dieser Vorlesung wird class immer benutzt für Objekt Orientierte programmierung!

Klassengerüst

```
int s = 10 ;// Globaler Variable
class Klassenname{
        private:
                Datentyp name;                          //Instanzvariable
                Static Datentyp name;                   //Klassenvariable
        public:
                Klassenname();                          //Default Konstructor
                Klassenname(Datentyp parameter); //Konstruktor
                Datentyp Funktion1();                   //Memberfunktion
                Datentyp Funktion2(Datentyp parameter);
                Datentyp Funktion3() const;             // const-Memberfunktion
                static Datentyp Funktion4()             //static-Memberfunktion
}; // Typisch für C++
```

- **Diese Klasse wird später über Main instanziiert.**

Klassengerüst mit Definition

Variante 1

```
class Klassenname{

//Deklaration
Datentyp Funktion2(Datentyp parameter);
};



//Definition
Datentyp Funktion2(Datentyp parameter)
{
Anweisung
}
```

Variante 2

```
class Klassenname{

//Gleichzeitig Deklaration und Definition

        Datentyp Funktion2(Datentyp parameter)
        {
        Anweisung
        }
};
```

Klassengerüst mit Headerdatei

Variante 1- Nicht empfehlungswert

- Headerdatei mit Definition!!!

```
//Headerdatei mit .h
class Klassenname{
//Deklaration
Datentyp Funktion2(Datentyp parameter);
//Definition
Datentyp Funktion2(Datentyp parameter)
{
Anweisung
}
};


//Nur Main in C++ Quelldatei
int Main()
{
Return 0;
}
```

Variante 2- Wird empfohlen

- Headerdatei mit Deklaration

```
//Headerdatei mit .h
//Deklaration
Datentyp Funktion2(Datentyp parameter);
class Klassenname{
//Definition und Main in C++ Quelldatei
Datentyp Funktion2(Datentyp parameter)
{
Anweisung
}
};

int Main()
{
Return 0;
}
```

Klassengerüst weiter...

Die Basis der OOP- Zusammengefasst

- C++ erlaubt unter anderen die Deklaration von Klassen und Strukturen zur Strukturierung eines OOP-Programms.
- Klassen und Strukturen unterscheiden sich in C++ prinzipiell kaum. In beiden können Sie Eigenschaften und Methoden unterbringen.
- Klassen werden mit dem Schlüsselwort class deklariert, Strukturen mit struct.
- Alle Datenfelder in einer Struktur sind öffentlich. Klassen ermöglichen dagegen, dass Datenfelder privat oder geschützt (protected) deklariert werden. Strukturen ermöglichen somit nicht das wichtige Konzept der **Kapselung.**

```
class person{
        string Name;  // private
        string Nachname   //private
      public:
        void laufen(){       //public
        }
};
```

```
struct  person{
        string Name;  // public
        string Nachname   //public
        void laufen(){       //public
        }
};
```

Die Strukturierung einer Anwendung

- Größere Programme erfordern eine Strukturierung des Quellcodes. Die Basis der Strukturierung ist in C++ eine Klasse.
- Über Klassen erreichen Sie, dass Sie Programmcode wiederverwenden können und dass Sie Ihr Programm so strukturieren, dass die Fehlersuche und die Wartung erheblich erleichtert werden.
- Wenn Sie eine Klasse entwickeln, können Sie entscheiden, ob Sie eine **echte** Klasse (mit **normalen** Eigenschaften und Methoden) programmieren, aus der Sie später Instanzen erzeugen, oder ob Sie eine Klasse mit statischen Methoden und Eigenschaften erzeugen wollen.
- Statische Methoden und Eigenschaften (Klassenmethoden, Klasseneigenschaften) können Sie auch ohne eine Instanz der Klasse aufrufen.
- **Echte** Klassen arbeiten echt objektorientiert, Klassen mit statischen Methoden und Eigenschaften simulieren (u. a.) die Module der strukturierten Programmierung.

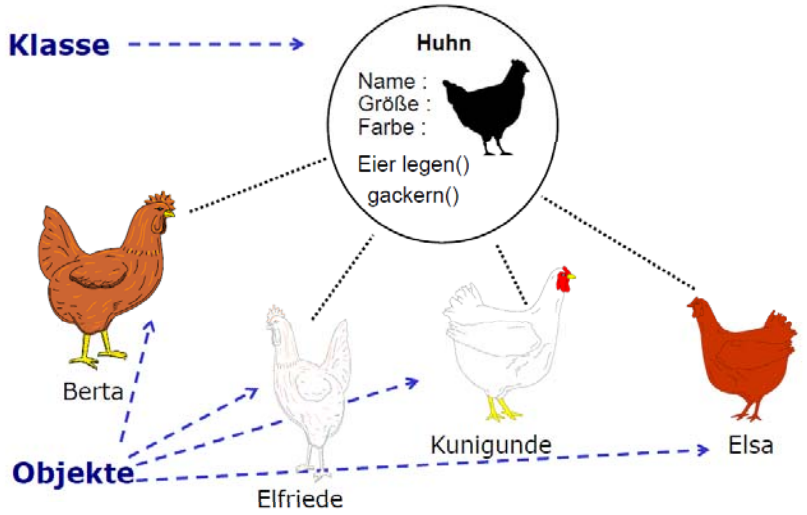
Einfache Klassen und deren Anwendung

Grundlagen zur Programmierung von Klassen

- In einer C++-Datei (mit der Endung *.cpp* oder *.cc*) können Sie eine oder mehrere Klassen implementieren.
- Sie können Klassen komplett in der *cpp*-Datei unterbringen und auf eine Headerdatei verzichten oder
- In *h*-Datei und *cpp*-Datei trennen. Die *h*-Datei (Headerdatei) enthält die Deklaration der Klasse, die *cpp*-Datei enthält die Implementierung der Methoden der Klasse.
- In einer *cpp*-Datei können Sie eine oder mehrere Klassen unterbringen.
- In der Praxis werden Klassen oft in separaten Dateien deklariert, um diese einfach in anderen Programmen wiederverwenden zu können.
- Diese Trennung macht auch dann Sinn, wenn Sie Ihre Klassen in Bibliotheken (mit der Endung *.lib*) kompilieren und diese Bibliotheken für die eigene Verwendung in einem separaten Ordner speichern.

Klassen üben

- **Schreiben Sie eine Klasse „Huhn“ und erzeugen Sie die 4 Hühner**



Konstruktoren

- Sie tragen die gleichen Namen wie die Klasse, in der sie definiert sind.
- Sie steuern die Erstellung/Erzeugung eines Objektes über die Main Methode. Es sind spezielle Methoden, die bei der Instanziierung von Objekten (Erzeugen eines Objektes der Klasse – s.u.) aufgerufen werden.
- Konstruktoren ohne Parameter werden als Default- oder Standardkonstruktoren bezeichnet.
- Wenn in einer Klasse explizit kein Konstruktor definiert wird, definiert der Compiler implizit eine Defaultkonstruktor.
- Beispieldefinition:

```
class Klassenname{
public:
        Klassenname(){
        //Definition hier
        }
};
```

Destruktoren

- Destruktoren: Nachdem das Objektes über Konstruktor-Aufruf erzeugt wurde, kann beliebig mit dem Objekt gearbeitet werden. Wird das Objekt nicht mehr benötigt, kann es über den Destruktor „zerstört“ werden. Dadurch wird der belegte Speicherplatz wieder frei.
- Es gibt auch einen Default/Standard Destruktor. Wird immer implizit aufgerufen.
- Rufen Sie nie ein Standard-Destruktor explizit auf!
- Destruktoren kann man auch explizit definieren.
- Hat auch immer den gleichen Namen wie der Klassenname.
- Beispieldefinition:

```
~Klassenname(){
//Definition hier
}
```

Siehe Beispiele...

Huhn-Klasse

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

class Huhn{

private:
    string name;
    int größe;
    string farbe;
    bool istHahn;
public:
    Huhn(string name, int größe, string farbe, bool istHahn){
    this->name = name;
    this->größe = größe;
    this->farbe = farbe;
 this->istHahn = istHahn;
    }
    bool EierLegen();
    bool Gackern();
 bool Huhn_ausgeben();

};
```

```
bool Huhn::EierLegen(){
if(istHahn) {
   return  false;
   }else{
   return  true;
}
}

bool Huhn::Gackern(){
if(istHahn) {
   return  false;
   }else{
   return  true;
}

}
```

Verwendung in der Main Methode

```
int main(){

Huhn  huhn1("Elsa", 2, "rot", false);    // Objekt Elsa ist erzeugt
Huhn  huhn2("Elfriede", 2, "weiß", false); // Objekt Elfriede ist erzeugt
Huhn  huhn3("Kunigunde", 3, "weiß", false); // Objekt Kunigunde ist erzeugt
Huhn  huhn4("Berta", 4, "braun", false); // Objekt Berta ist erzeugt

cout<< "Elsa legt Eier:"<<huhn1.Eierlegen() << endl;
cout<< "Elsa Gackert:"<<huhn1.Gackern() << endl;
cout<< "Elfriede legt Eier:"<<huhn2.Eierlegen()<< endl;
cout<< "Elfriede Gackert:"<<huhn2.Gackern() << endl;

cin.get();
return 0;
}
```

Zugriffsrechte

- Alle public und protected Methoden(Funktionen) und Variablen(Eigenschaften) werden vererbt und sind dadurch auch in der Subklasse vorhanden.
- Diese Sichtbarkeit kann mit dem Zugriffsrecht nach dem Doppelpunkt weiter eingeschränkt werden:

Variable/Methode in Superklasse	Subklasse vererbt als		
	public	protected	private (Def.)
	Variable/Methode in Subklasse		
public	public	protected	private
protected	protected	protected	private
private	nicht vererbt	nicht vererbt	nicht vererbt

- Man beachte, dass der Default Zugriffsrecht **private** ist und nicht **public** – auch wenn man in den allermeisten Fällen public benötigt.

Zugriffrecht

- **Beispiel zu private / protected / public Vererbung:**

```
class A {
private:
    int priv;
protected:
    int prot;
public:
    int publ;
};
```

```
class B : pxxx  A {
    public:
        void m() {...};
};

void f() {
A a;
B b;
}
```

Zugriff innerhalb B (m())	unabhängig von pxxx
priv++;	nein
prot++;	ja
publ++;	ja

Zugriff von aussen (f())	pxxx = private	pxxx = protected	pxxx = public
a.priv++;	nein	nein	nein
a.prot++;	nein	nein	nein
a.publ++;	ja	ja	ja
b.priv++;	nein	nein	nein
b.prot++;	nein	nein	nein
b.publ++;	nein	nein	ja

Zugriffrecht

- private / protected Vererbung: alle public Methoden der Superklasse sind in der Subklasse nicht mehr public. Oft eine zu starke Einschränkung.
- Generell wird private/protected Vererbung in der Praxis selten verwendet. Mit sauberem OODesign sollten Sie private/protected Vererbung eigentlich nicht verwenden müssen.

Zugriffrecht(friend)

Friend Klassen:

- In einigen Fällen kann es notwendig werden, dass andere Klassen oder Funktionen Zugriff auf die geschützten Member einer Klasse benötigen.
- Damit eine Klasse *Class1* Zugriff auf alle Member einer anderen Klasse *Class2* erhält, wird die Klasse *Class1* als *friend*-Klasse der Klasse *Class2* deklariert. Dazu wird innerhalb der Klassendefinition der Klasse, die ihren 'Schutz' aufgibt, folgende Anweisung eingefügt:

  Siehe Beispiel : friend_to_friend

Zugriffrecht(friend)

Friend methoden:

- Außer den *friend*-Klassen gibt es auch noch *friend*-Methoden. *friend*-Methoden gehören keiner Klasse an und haben ebenfalls vollen Zugriff auf alle Member einer Klasse. Um eine Methode als *friend*-Methode einer Klasse zu deklarieren, wird wieder innerhalb der Klasse die ihren 'Schutz' aufgibt folgende Anweisung eingefügt:

  friend Rückgabetyp Methodenname(...);

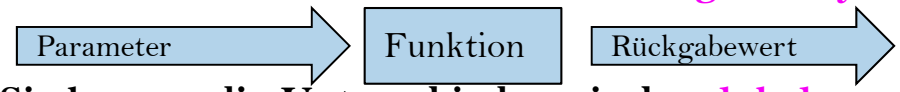
- *Man beachte:*

  Die *friend*-Eigenschaft einer Klasse ist nicht vererbbar.

- *Siehe Beispiel: friendfunction*

Ziele für Methoden

- **Sie wissen was Funktionen in C/C++ sind, wozu sie gebraucht werden können und wie sie definiert und deklariert werden.**
- **Sie verstehen das Prinzip von Rückgabewerten und Parametern; insbesondere verstehen Sie, wie die Parameterübergabe „by value“ genau funktioniert.**



- **Sie kennen die Unterschiede zwischen lokalen, globalen, Klassen und statischen lokalen Variablen und können das Konzept der Sichtbarkeit von lokalen Variablen erklären.**
- **Sie kennen das Konzept der Parameterübergabe „by address“ („by reference“) und können es richtig anwenden.**
- **Sie wissen, wie Arrays einer Funktion übergeben werden und wie Pointer als Rückgabewert verwendet können.**
- **Sie wissen, was konstante Parameter sind und welchen Einfluss sie haben.**
- **Sie wissen, wie man einem C/C++-Programm Kommandozeilen Parameter übergibt.**

Funktionen/Methoden in C++

- **Funktionen sind ganz zentrale Komponenten von C/C++-Programmen:**
- **Erlaubt die Strukturierung von Programmen**
- **Programmteile, die mehrmals in einem Programm vorkommen, müssen nur einmal implementieren werden**
- **In C++ werden die meistens Methoden genannt, und wir werden ab jetzt nur Methoden nennen.**
- **Auch main, der Einstiegspunkt eines C++-Programms, ist nichts anderes als eine spezielle Methode**
- **Methoden bestehen aus:**
  - Methodenname (identifiziert die Methode eindeutig)
  - Parameterliste (mehrere Parameter mit Komma getrennt; leer, wenn keine Parameter übergeben werden). void bei leeren Parameterlisten ist nicht mehr nötig. Der Compiler prüft die übergegeben Parameter in jedem Fall. void kann aber immer noch verwendet werden, wir lassen es hier weg
  - Datentyp des Rückgabewerts (void, wenn nichts zurückgegeben wird)

Methoden in C++

- **Methode**

  Datentyp_des_Rückgabewerts Methodenname(Parameterliste)

- **Ein Parameter besteht aus dem Typ(Datentyp) und einem Namen**
  - Mit dem Namen kann der Parameter innerhalb der Methode angesprochen werden
- **Beispiel:** **int max(int a, int b)**
  - Eine Methode mit dem Namen max
  - Die Methode hat zwei Parameter vom Typ int, die Namen sind a und b
  - Die Methode gibt einen int-Wert zurück
- **Alle Methoden in einem C++-Programm müssen nicht wie in C verschiedene Namen haben! Methoden könne überladen oder überschrieben werden**

int max(int a, int b)

int max(int a, int b, int c)

long max(long a)

void max()

int max()

Methoden Deklaration und Definition

- **Deklaration**

```
int max(int a, int b);
```

Definition

```
int max(int a, int b) {
if(a >= b) {
return a;
}
return b;
}
```

- **Typischerweise wird die Methodendeklaration in eine Header Datei (.h Datei) gemacht. Wie in der letzten Vorlesungen gesehen, man kann auch bei der Klassendefinition direkt nach der Varaibeln eingefügt werden.**

Aufruf von Methoden

- **Beim Methodenaufruf müssen für sämtliche Parameter Werte übergeben werden**
  - Meist mittels Variablen; Übergabe von Literalen ist auch möglich
  - Reihenfolge entspricht der Parameterliste in der Funktionsdefinition
- **Hat die Methode einen Rückgabewert, so kann (muss aber nicht) dieser einer Variablen zugewiesen werden oder gleich in einem Ausdruck wiederverwendet werden**
- **Beispiel:**

```
int a = 5, b = 3, c;
c = max(a, b);                          /* c = 5 */
c = max(7, 13);                         /* c = 13 */
max(15, 4);                             /* OK, aber hier
                                        sinnlos */
c = 5 + max(8, a);                      /* c = 13 */
c = max(max(5, 9), max (b, 12));        /* c = 12 */
```

Parameter und Rückgabewerte

- **Folgende Typen sind als Parameter zugelassen:**
  - Grundlegende Datentypen (int, double…)
  - Strukturen
  - Arrays
  - Pointer
- **Folgende Typen sind als Rückgabewerte zugelassen:**
  - Grundlegende Datentypen (int, double…)
  - Strukturen
  - Pointer
- **Es können keine Arrays zurückgegeben werden, aber Pointer auf Datenbereiche, die einen Array enthalten!**
- **Mit return wird eine Methode immer sofort verlassen (wie Java)**
  - Hat die Methode keinen Rückgabewert (void), so wird einfach return; ohne Rückgabewert geschrieben
  - Hat die Methode einen Rückgabewert, so wird return mit einem entsprechenden Wert geschrieben, z.B. return -1;

Parameter und Rückgabewerte

- **Wird das Ende einer Methode erreicht, so wird dieMethode auch ohne Angabe von return verlassen**
  - Hat die Methode einen Rückgabewert, so wird per Default der Wert 0 zurückgegeben (besser: explizit Wert angeben)
- **Die Rückgabe eines Wertes erfolgt „by value“**
- **main: verwendet der Programmierer am Ende der Methode kein return, so fügt der Compiler auch hier ein return 0 ein**
  - Rückgabewert von main wird an die aufrufende Umgebung (zB Shell oder irgendein anderes Programm) zurückgegeben
  - Auch aus main kann jederzeit ein beliebiger Wert zurückgegeben und damit das Programm terminiert werden

Lokale Variablen

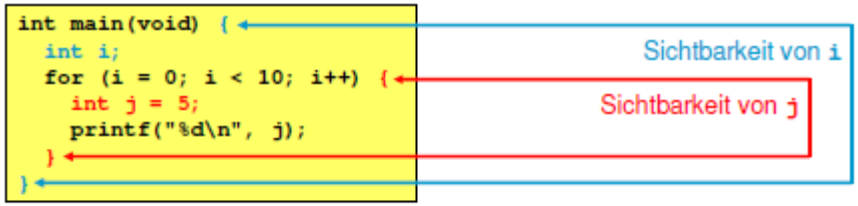
- **Neben den Parametern, welche einer Methode übergeben werden, können lokale Variablen definiert werden**

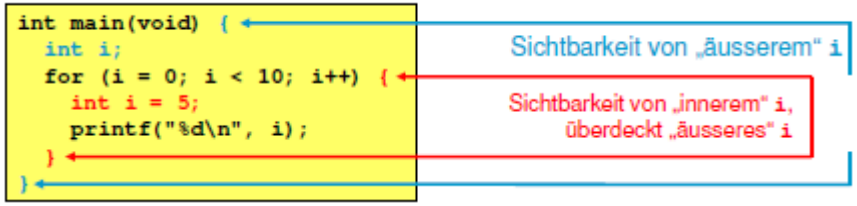
```
void printLine(int n) {
int i;                                  /* Lokale Variable */
for (i = 0; i < n; i++) {
cout<<"_";
}
cout<<endl;
}
```

- **Die lokalen Variablen sind nur innerhalb der Methode sichtbar**
- **Es entsteht kein Konflikt mit einer Variablen in einer anderen Methode, die den gleichen Namen hat**
- **Die Variablen in der main Methode sind auch lokale Variablen**
- **Alle innerhalb von Methoden deklarierten Variablen sind lokale Variablen**

Sichtbarkeit von Lokale Variablen

- **Generell sind lokale Variablen nur in dem Block sichtbar (verwendbar), in welchem sie deklariert wurden**





Klassen Variablen

- **Neben lokalen Variablen innerhalb von Methoden gibt es auch globale Variablen, auf die alle Methoden zugreifen können.**
- **Globale Variablen werden außerhalb von allen Methoden definiert und müssen in jeder Methode, die auf sie zugreift, mit extern deklariert werden**
  - Die Angabe der globalen Variablen mit „extern“ innerhalb von Methoden, wo sie verwendet wird, hat den gleichen Zweck wie eine Methodendeklaration: Der Compiler erhält die Information, dass es die globale Variable gibt (sie könnte ja in einer anderen Datei definiert sein) und kennt ihren Datentyp.
- **Mit globalen Variablen sollte sparsam umgegangen werden!**

```
/* checkmax_glob.c */
#include <iostream>
 using namespace std;
 int max = 0;            /* Definition glob. Variable */
 void checkMax(int a);

int main(void) {
  int a;
  extern int max;    /* Deklaration glob. Variable */
  while (1) {
  Cout<<a = „;
  ...
  checkMax(a);
  Cout<<max;
   }
}
void checkMax(int a) {
  extern int max;    /* Deklaration glob. Variable */
  if (a > max) {
  max = a;
  }
}
```

Klassen Variablen

- **Auf die Klassen Variablen können nur alle Methoden in ein Klasse oder Vererbte Klassen Methoden zugreifen. Hängt sehr eng zusammen mit Zugriffsrechte.**

Statische Lokale Variablen

- Lokale Variablen, die mit static als statische Variablen deklariert werden, behalten ihren Wert auch nach dem Verlassen und Wiedereintritt in eine Funktion
- Statische Variablen werden nur einmal, gleich nach dem Programmstart initialisiert
- Statische Variablen erhalten per Default den Wert 0
- Oft lassen sich damit globale Variablen ersetzen

```
/* checkmax_stat.c */
#include <iostream>
using namepace std;
int checkMax(int a);
int main(void) {
int a;
while (1) {
Cout<<a = ";
...
Cout<<checkMax(a);
}
}
int checkMax(int a) {
static int max = 0;          /* Deklaration */
if (a > max) {               /* statische */
max = a;                     /* lokale Variable */
}
return max
}
```

Rekursive Methoden

- **C++ unterstützt rekursive Methoden genau so wie C: Eine methode kann sich selbst wieder aufrufen**
- **Bei jedem weiteren rekursiven Aufruf werden Parameter wieder „by value“ übergeben, ebenfalls hat jede Rekursionsstufe eigene lokale Variablen**
- **Viele Probleme lassen sich mit Rekursion kompakt und elegant lösen**
- **Beispiel: Fakultät einer Zahl (n! = n * n-1 *...* 1, 0! = 1)**

```
/* Iterativ */

int fakultaet(int n) {
  int i = 1;
  int res = 1;
  for (i = 2; i <= n; i++) {
  res = res * i;
  }
return res;
}
```

```
/* Rekursiv */

int fakultaet(int n) {
  if (n < 2) {
  return 1;
  } else {
  return n * fakultaet(n – 1);
  }
}
```