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

Heute

- **Dynamische Speicherverwaltung**
  - Was ist dynamische Speicherverwaltung
  - Wie geht man mit **New** und **Delete** um
- **Methoden in C++**
- **Vererbung**

Dynamische Speicherverwaltung

Wir werden uns mit der Möglichkeit vertraut machen, Speicher für Objekte erst während des Ablaufs des Programms (Laufzeit) zu reservieren. Dabei geht es um folgende Fragen:

- **Was ist dynamische Speicherverwaltung:**
  - Wir müssen als Programmierer unsere Programme flexibel gestalten. Das bedeutet, dass wir nicht schon zum Zeitpunkt der Erstellung festlegen dürfen können, wie viele Objekte jemals bearbeitet werden.
  - Das Programm selbst in der Lage sein, dynamisch auf die aktuellen Anforderungen zu reagieren.
  - Und das kann erst zur Laufzeit des Programms erfolgen, dazu benötigen wir eine dynamische Speicherverwaltung.
- Schauen wir erst einmal zurück. Zwei Formen der Speicherverwaltung:
  - Die statische Speicherverwaltung. Statisch bedeutet in diesem Fall, dass die benötigte Speichermenge bereits bei der Erstellung des Programms bekannt ist.
  - Auch eine Form der dynamischen Speicherverwaltung kennen wir schon: Lokale Variable werden erst zur Laufzeit des Programms angelegt. Allerdings entzieht sich deren Verwaltung unserer Kontrolle und auch die benötigte Speichermenge muss bereits bei der Erstellung des Programms bekannt sein.

Dynamische Speicherverwaltung

Wie geht man mit New und Delete um

- Der Operator **new** ist ein direkter Bestandteil der Sprache C++ und damit ein Schlüsselwort. Mit diesem Operator können wir Speicher während des Programmlaufs anfordern, wenn wir ihn brauchen.

```
Konto *  ptrKto;  // Zeiger auf Konto

ptrKto = new Konto;                // 1.
ptrKto = new (Konto);              // wie 1.
ptrKto = new Konto (32168, 0.0);   // 2.
ptrKto = new Konto [Anzahl];       // 3.
```

Wirkung:

- 1. new reserviert Speicherplatz für ein Objekt vom Typ Konto und initialisiert dieses Objekt durch Aufruf des Default-Konstruktors (= Konstruktor ohne Argumente).
- 2. Hier wird ebenfalls ein Objekt vom Typ Konto angelegt. Aber es wird der Konstruktor mit passender Initialisierungsliste aufgerufen (KtoNr = 32168, Stand = 0.0).
- 3. new reserviert hier Speicherplatz für eine bestimmte Anzahl an Konten. Anzahl darf eine Variable eines Integer-Typs sein. Es kann hier nur der Default-Konstruktor zur Initialisierung der Konten angewandt werden. Er wird automatisch für jedes Element des Feldes aufgerufen.

Dynamische Speicherverwaltung

Wie geht man mit New und Delete um

```
Konto *  ptrKto1;  // Zeiger auf ein Konto
Konto *  ptrKtoN;  // Zeiger auf N Konten

ptrKto1 = new Konto(32168, 0.0);
ptrKtoN = new Konto [Anzahl];

delete ptrKto1;     // 1.
delete [] ptrKtoN;  // 2.
```

Wirkung:

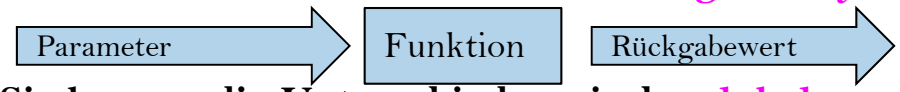
- 1. delete gibt den Speicherplatz für das eine mit new reservierte Objekt wieder frei. Vorab wird der Destruktor des Objekts aufgerufen.
- 2. Hier wird das gesamte reservierte Feld wieder freigegeben. Für jedes einzelne Element des Feldes wird der Destruktor aufgerufen.

Methoden in C++

- **Was wissen Sie schon?**
- **Siehe Beispiel…**
- **Was müssen Sie wissen?**

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 und auch vorhin gesehen, man kann die auch bei der Klassendefinition direkt nach der Variablen einfügen.**

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…) und benutzer definierte Datentypen.
  - Strukturen
  - Arrays
  - Pointer
- **Folgende Typen sind als Rückgabewerte zugelassen:**
  - Grundlegende Datentypen (int, double…), auch benutzer definierte Datentypen
  - 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 in 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 die Methode 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

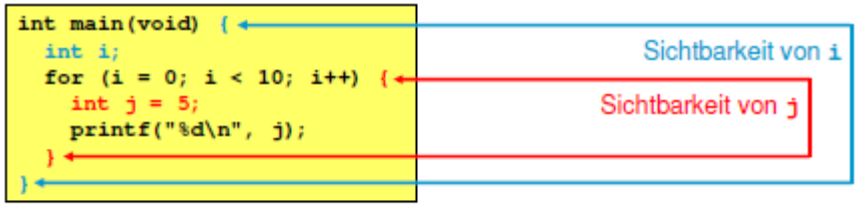
- **Neben den Parametern, welche einer Methode übergeben werden, können lokale Variablen definiert werden**

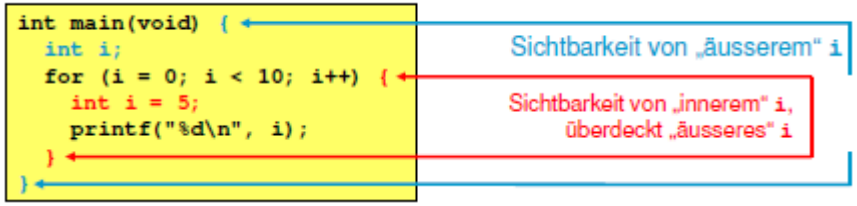
```
void printLine(int n) {
int i;                                  /* Lokale Variable */
for (i = 0; i < n; i++) {
cout<<"_";
}
cout<<endl;
}
```

- **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
}
```

Call by Value

- **Sie haben im 1. Teil über Funktionen erfahren, dass Parameter immer „by value“ einer methode übergeben werden**
  - Die aktuellen Werte der Variablen werden in die Parameter der Methode kopiert
  - Ein Ändern dieser Werte innerhalb der Methode hat keinen Einfluss auf die Werte der Variablen, die der Methode übergeben worden sind
- **Call by value macht aber nicht in allen Fällen Sinn, z.B.:**
  - Kopieren ist bei grossen Datenstrukturen (Arrays, Strukturen) ineffizient
  - Die Methodesoll mehr als einen Wert zurückgeben
- **Deshalb kann in C++ als auch in C nicht nur der Wert einer Variablen, sondern auch die Adresse einer Variablen einer Methode übergeben werden dazu übergibt man den Pointer auf diese Variable**
- **In diesem Fall spricht man auch von „call by address“, auch wenn die Adresse (der aktuelle Wert des Pointers) selbst immer noch „by value“ übergeben wird**

Call by Adress

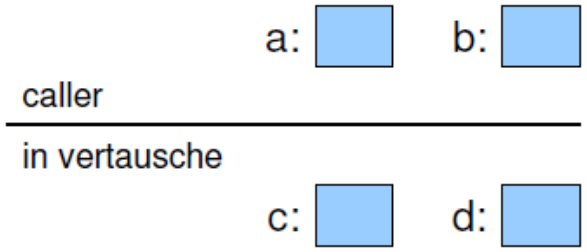
- **Beispiel: Methode vertausche, die die Werte zweier Variablen vertauschen soll**

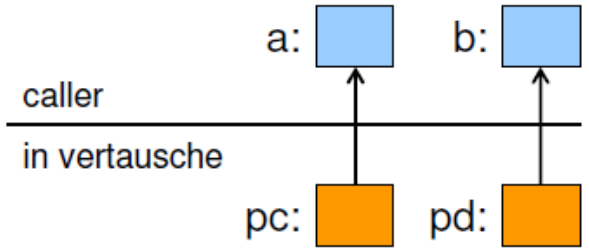
Call by value (funktioniert nicht):

```
int a = 3, b = 5;
vertausche(a, b);

void vertausche(int c, int d) {
  int temp = c;
  c = d;
  d = temp;
}
```

Call by address (funktioniert):

```
int a = 3, b = 5;
vertausche(&a, &b);

void vertausche(int *pc, int *pd) {
  int temp = *pc;
  *pc = *pd;
  *pd = temp;
}
```





Arrays als Parameter

- **Wird ein Array in einem Ausdruck verwendet, so wird er implizit in den entsprechenden Pointer konvertiert (siehe Arrays & Pointer)**
- **Wird ein Array einer Methode übergeben, wird implizit ein Pointer auf diesen Array übergeben -> Arrays werden immer „by address“ übergeben**
- **Weiter spielt es keine Rolle, ob einer Methode vom Programmierer explizit ein Array oder ein Pointer auf einen Array übergeben wird; für die Methode ist es immer ein Pointer**
- **Ebenso kann in der Parameterliste ein Parameter entweder als Array oder als Pointer spezifiziert werden:**
  - rückgabewert funcName(int a[]);
  - rückgabewert funcName(int *a);
- **Beachte: wird der Parameter als 1-Dimensionaler Array spezifiziert, dann wird die Dimension des Arrays in der Parameterliste nicht spezifiziert (den mehrdimensionalen Fall betrachten wir später)**

Arrays als Parameter

- **Beispiel: beide Methoden sind äquivalent (sie berechnen das Skalarprodukt zweier Arrays):**

```
double sprod(double first[],
             double second[],
             int len) {
  double sum = 0.0;
  int i;

  for (i = 0; i < len; i++) {
    sum += first[i] *
           second[i];
  }
  return sum;
}
```

können auch vertauscht werden!

```
double sprod(double *first,
             double *second,
             int len) {
  double sum = 0.0;
  int i;

  for (i = 0; i < len; i++) {
    sum += *first++ *
           *second++;
  }
  return sum;
}
```

- **Die Parameterübergabe beim Methodenaufruf kann ebenfalls mit Arrays oder mit Pointern erfolgen – unabhängig davon, welche der beiden Methoden oben implementiert wurde**

Array als Rückgabewert?

- **Der Rückgabewert einer Methode kann kein Array sein, sehr wohl aber ein Pointer auf einen Array**
- **Beispiel: Methode, die drei int-Parameter erhält, daraus einen int-Array der Länge *3* bildet und einen Pointer auf diesen Array zurückgibt:**

```
/* getarray1.c */
int *getArray(int a, int b,
              int c) {
  int ret[3];

  ret[0] = a;
  ret[1] = b;
  ret[2] = c;
  return ret;
}
```

Leider funktioniert das nicht – wieso? (kein Kompilierfehler!)

So funktioniert's:

```
/* getarray2.c */
int *getArray(int a, int b,
              int c) {
  int *ret = (int *) malloc(
          3 * sizeof(int));

  *ret = a;
  *(ret + 1) = b;
  *(ret + 2) = c;
  return ret;
}
```

- **Generell kann eine Methode einen Pointer auf irgendetwas zurückgeben**

Konstante Parameter

- **Ein Array wird immer by address einer Funktion übergeben; der Programmierer kann nicht wählen, den Array by value zu übergeben**
  - Es gibt keine lokale Kopie des Arrays innerhalb der Funktion
  - Innerhalb der Funktion wird mit dem gleichen Array gearbeitet, wie
  - ausserhalb (Zugriff auf die gleichen Speicherstellen)

- **Dadurch hat jede Modifikation der Arrayelemente in der Funktion direkten Einfluss auf den Array, welcher der Funktion „übergeben“ wurde**
  - Potentielles Problem: wie kann der Programmierer z.B. sicher sein, dass
  - eine Funktion einer Library den Array nicht modifiziert?

- **Mittels Spezifizieren eines Parameters mit const stellt der Compiler sicher, dass der entsprechende Parameter innerhalb der Funktion nicht modifiziert wird!**

Konstante Parameter

- **Beispiel: Methode, die eine Zahl, die als String dargestellt ist, in einen int konvertiert; der String darf in der methode nicht verändert werden!**
- **Der Gebrauch von const in der Parameterliste (wenn sinnvoll) ist guter Programmierstil, gewöhnen Sie sich dies an!**

Parameter als konstanter char-Array

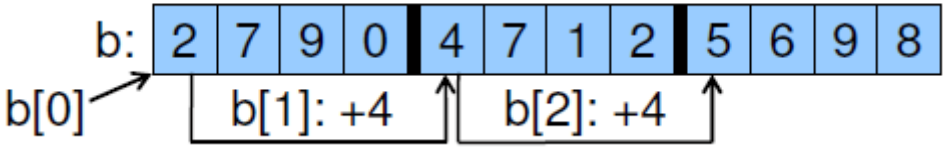
```
int stringToInt(const char s[]) {
  int i, res = 0;

  for (i = 0; s[i] != '\0'; i++) {
    res = res * 10 + s[i] - '0';
  }
  return res;
}
```

Parameter als Pointer auf einen konstanten char

```
int stringToInt(const char* s) {
  int res = 0;

  for (; *s != '\0'; s++) {
    res = res * 10 + *s - '0';
  }
  return res;
}
```

2–Dimensionale Arrays als Parameter

- **Wird ein 2-Dimensionaler Array a[3][4] einer Mthode übergeben, so muss in der Parameterliste die zweite Dimension spezifiziert werden!**
  - rückgabewert funcName(int b[][4]);
  - rückgabewert funcName(int b[][]); /* FALSCH, Kompilierfehler! */
- **Die zweite Dimension ist notwendig, damit in der Methode Anweisungen wie b[2] korrekt ausführt werden können:**
  - Die Angabe der zweiten Dimension ermöglicht es herauszufinden, wo die einzelnen Arrays im Array beginnen (wegen der linearen Anordnung im Speicher)



  - Ohne die Angabe dieser zweiten Dimension wüsste der Compiler nicht, wie auf die einzelnen Arrays zugegriffen wird, denn bei der Parameterübergabe wird nur ein Pointer auf b übergeben

Array of Pointers als Parameter

- **Wird ein Array of Pointers, z.B. int *a[10] einer Methode übergeben, ist dies nichts anderes als die Übergabe eines 1-Dimensionalen Arrays**
  - rückgabewert funcName(int *b[]);
- **Bei der Konvertierung in einen Pointer wird der Datentyp zu int **b (Pointer auf einen Pointer auf einen int, siehe Vorlesung über Arrays und Pointer)**
- **Deshalb ist diese Funktionsdeklaration äquivalent zu der oben stehenden:**
  - rückgabewert funcName(int **b);
- **Wiederum können unabhängig davon, welche Methodendeklaration verwendet wird, Variablen der Typen int*[10] und int** übergeben werden**

Methoden als Parameter

- **Man kann einer Methode auch einen Pointer auf eine methode übergeben!**
- **Beispiel: Integration mit Trapezregel für beliebige Methoden**

```
/* trapez.c */



double trapez (double (*func) (double x), double start, double end, int n);

double trapez (double (*func) (double x), double start, double end, int n) {
  double sum = 0.0;
  double t = start;
  double interval = (end - start) / n;
  int i;
  for (i = 0 ; i < n ; i++) {
    sum += ((*func)(t) + (*func)(t+interval))/2 * interval;
    t += interval;
  }
  return sum;
}

int main(void) {
  double pi = 3.1415926;
  double integral = trapez(sin, 0, pi/2, 100);

}
```

Bedeutet: Pointer auf eine Funktion, die einen Parameter vom Typ double erhält und einen Wert vom Typ double zurückgibt

Gebrauch der übergebenen Funktion

In `math.h`:
`double sin(double x);`

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;
}
```

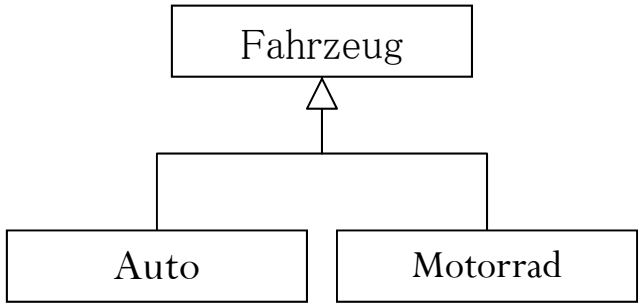
```
/* Rekursiv */

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

Einführung in der Vererbung

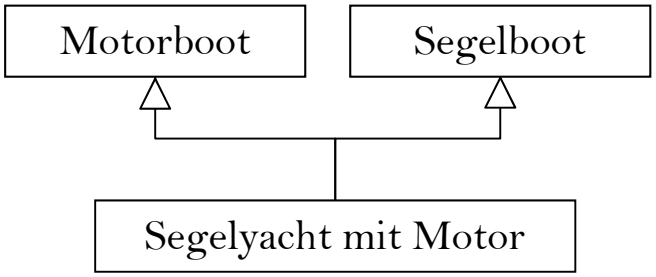
- Die Vererbung ermöglicht es, neue Klassen auf der Basis von schon bestehenden Klassen zu definieren.
- In C++ ist es möglich eine neue Klasse von mehreren Basisklassen abzuleiten, die dann die Eigenschaften aller dieser Basisklasssen besitzt. Dazu kommen all die Eigenschaften, die man neu definiert.
- Vererbung – Einbettung: Um in einer Klasse die Eigenschaften einer anderen nutzen zu können, gibt es neben der Vererbung und der friend- Deklaration die Möglichkeit der Einbettung.
- Die Entscheidung ob Vererbung oder Einbettung sinnvoller ist, hängt von der Beziehung ab, in der die Objekte zueinander stehen

Einführung in der Vererbung

Einfachvererbung



Mehrfachvererbung

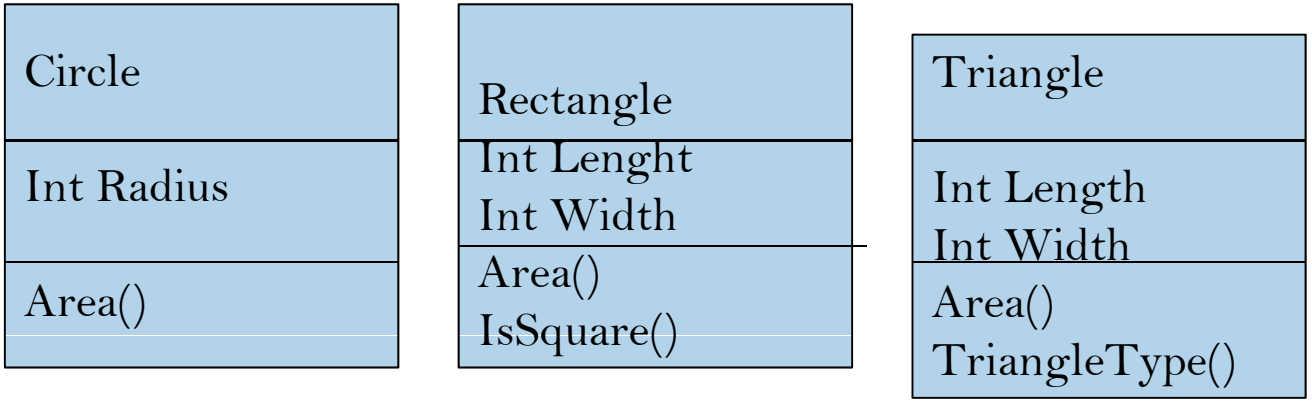


→ Alle Attribute und Methoden sind in der erbenden Klasse enthalten

Einführung in der Vererbung

Warum Vererbung?

- Wir wollen mit Flächen rechnen. Wir stellen uns dazu folgende Klasen vor:



- Wenn wir diese Klassen realisieren, stellen wir fest, dass der Quelltext der Klassen weitgehend identisch sind.
- **Problem:** Redundanz(Ein beträchtlicher Teil des Codes ist redundant)

Einführung in der Vererbung

- **Nachteile dieses Ansatzes:**
  - erhöhter Aufwand der Erstellung
  - bei der Wartung von dupliziertem Code sind Änderungen an mehreren Stellen notwendig
  - Wartung ist fehleranf ällig
  - erhöhter Testaufwand
  - erhöhter Aufwand bei der Nutzung der Klassen
- Statt die drei Klassen Circle, Rectangle und Triangle unabhängig voneinander zu definieren, definieren wir zuerst eine Klasse, die die Gemeinsamkeiten von allen zusammenfasst (z.B. Shape)- Prinzip der Generalisierung
- Wir definieren dann anschließend, dass ein Circle ein Shape ist und ebenso, dass ein Rectangle und ein Triangle ein Shape ist.
- Schließlich ergänzen wir die Klassen ausschließlich um ihre spezifischen Eigenschaften.

Einführung in der Vererbung

Bestandteile objektorientierter Programmierung:

- Bestehende Klassen können durch Spezialisierung und Erweiterung weiterentwickelt werden, ohne den Programmcode der alten Klassen verändern zu müssen

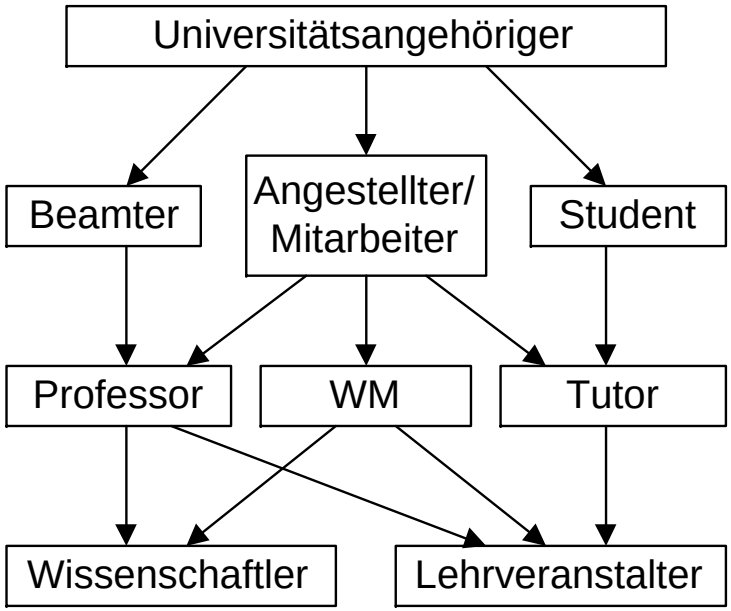
  ⇒ **Vererbung**

  - Dadurch erspart man sich zunächst einmal, bestehenden Programmcode neu zu schreiben oder zu duplizieren.
  - Statt dessen schreibt man eine neue Klasse und gibt bei der Beschreibung an, von welcher bestehenden Klasse ausgegangen werden soll (Ableiten aus dieser Klasse).
  - Die neue Klasse erbt Methoden und Attribute der alten Klasse (Basisklasse oder Superklasse) und kann zusätzlich durch neue ergänzt werden bzw. Methoden der alten Klasse können umdefiniert werden.

- Die Idee, Klassenhierarchien aufzubauen, sollte aber nicht allein dazu dienen, Schreibweisen zu vereinfachen, sondern konzeptionelle Ideen von Programmdesign umsetzen. Im wesentlichen bedeutet das für uns, dass die Sprachmechanismen von C++ dazu genutzt werden sollen, Gemeinsamkeiten zwischen Klassen auszudrücken.

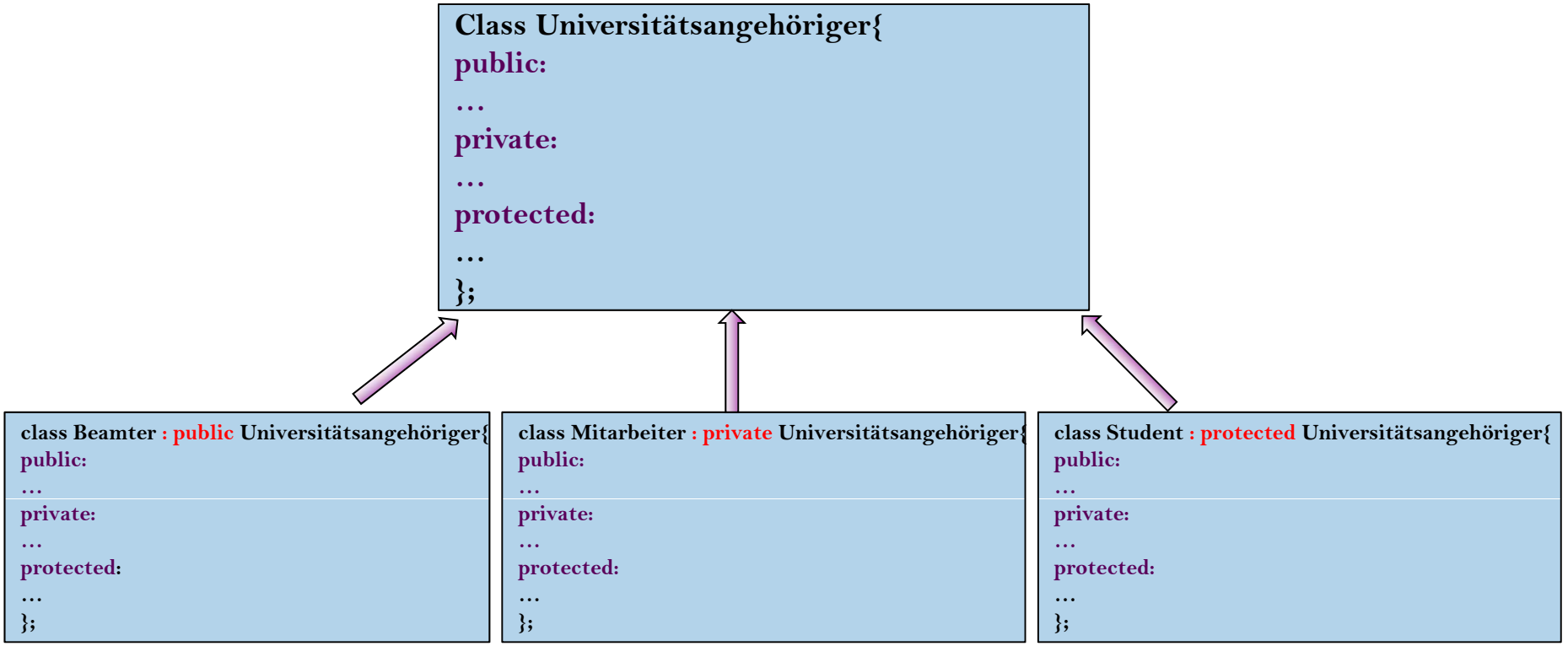
Einführung in der Vererbung

- Beispiel: Das Konzept von einem Beamten, einem Studenten und einem Angestellten / Mitarbeiter sind miteinander verwandt. Alle drei sind Universitätsangehöriger. Bei der Darstellung von Beamten, Studenten und Angestellter/Mitarbeitern in einem Programm, ohne die Notation einer Universitätsangehöriger einzuführen, fehlt „etwas“.



Einführung in der Vererbung

- Der Compiler muss von dieser Beziehung mitgeteilt werden. Ableiten einer Klasse mit Doppelpunkt : und Zugriffsrecht:



Mechanismus der Vererbung:

- Beamter, Student und Mitarbeiter sind von Universitätsangehöriger abgeleitet.
- Universitätsangehöriger ist Basisklasse für aller drei.
- Alle drei besitzen alle Member von Universitätsangehöriger plus zusätzlich eigenen.
- Alle drei können so auch überall dort verwendet werden, wo ein Universitätsangehöriger erlaubt ist.
- Das Objekt einer abgeleiteten Klasse kann deshalb wie ein Objekt der Basisklasse(Superklasse) behandelt werden, wenn es über Zeiger und Referenzen manipuliert wird. Die Umkehrung hiervon ist ungültig:

```
void Vorlesung(Student s, Universitätsangehöriger Ua )
{
Universitätsangehöriger * pUa = &s;      // o.k.
Student* ps = &Ua;                       // error
}
```

Einführung in der Vererbung

Terminologie

- Eine Superklasse (Oberklasse) ist eine Klasse, die von anderen Klassen erweitert wird.
- Eine Subklasse (Unterklasse) ist eine Klasse, die eine andere Klasse erweitert.
- Man sagt auch, dass die Subklasse von der Superklasse erbt.
- Vererbung bedeutet, dass die Subklasse alle Datenfelder und Methoden von der Superklasse übernimmt.
- Es werden keine Konstruktoren vererbt!
- Klassen, die über eine Vererbungsbeziehung miteinander verknüpft sind, bilden eine Vererbungshierarchie.

Einführung in der Vererbung

Vererbung und Zugriffsrechte

- Von einer Unterklasse aus kann man nicht auf die private deklarierten Datenfelder und Methoden der Oberklasse zugreifen.
- Daher gibt es zusätzlich den Modifikator (Zugriffsrecht) protected.
- Das Zugriffsrecht protected
  - erlaubt den Zugriff von Unterklassen aus,
  - erlaubt den Zugriff für Klassen des gleichen Pakets und
  - verbietet den Zugriff für alle anderen Klassen.
- Der Modifikator protected kann nur auf Datenfelder, Konstruktoren und Methoden angewendet werden, nicht auf Klassen.
- Datenfelder werden nicht als protected deklariert, da dies die Kapselung schwächen würde. Stattdessen definiert man Zugriffsmethoden die protected oder public sind.
- Typischer Einsatz von protected: Bei Hilfsmethoden, die nach außen verborgen werden sollen, für Unterklassen aber hilfreich sein können.

Vererbung - Syntax

- class klassenname :

  [virtual][public, protectet, private] Basisklassenname1, [virtual][public, protectet, private] Basisklassenname2

  {

  Liste der klassenelemente

  };
- Die Zugriffsspezifizierer public, protectet und private regeln bei der Vererbung nicht den Zugriff aus den Methoden der abgeleiteten Klassen.
- Virtual ist nur bei Mehrfachvererbung interessant.

Vererbung: Zugriffsbeschränkung

- Ausschlaggebend sind also die Zugriffsrechte, die in der Basisklasse vorgesehen waren, diese können dann in der abgeleiteten Klasse durch die Zugriffsspezifizierer nur verschärft 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