Funktionstemplates

- Einbinden von Templates:
  - Eine *.o Datei, die durch Übersetzen einer Datei nur mit Templates erzeugt wurde, enthält **keinen Programmcode und keine Daten**.
  - Templates sind keine Funktionsdefinitionen im üblichen Sinn, sondern eben Schablonen, nach der der Compiler **bei Bedarf** eine Funktion zu einem konkreten Datentyp erzeugt:

```
int main ( ) {
   vector<int>  vint;
   //  vint wird irgendwie gefüllt.
   mergesort(vint, ..., ...);

   vector<float> vfloat;
   // vfloat wird irgendwie gefüllt.
   mergesort(vfloat, ..., ....);
}
```

Compiler erzeugt ein erstes echtes Programm, in dem er „T“ in der Schablone durch „int“ ersetzt und das resultierende Programm übersetzt und einbindet.

Compiler erzeugt ein zweites echtes Programm, in dem er „T“ in der Schablone durch „float“ ersetzt und das resultierende Programm übersetzt und einbindet.

Breymann_Folien

Funktionstemplates

- In den Funktionstemplates kann man Variablen vom Typ T (<class T> oder <typename T>) deklarieren, definieren und man kann mit diesen Variablen arbeiten:

```
T  a, b, c;
a = b;                    //  Der Zuweisungsoperator muss in der
                          //  Klasse T definiert sein.
if (a < c)  funct1(a,c);  //  Der Vergleichsoperator muss in T
                          //  definiert sein und die Funktion funct1
                          //  mit den entsprechenden Parametern
                          //  muss definiert (implementiert) sein.
```

- Man beachte: Alle Operatoren und Funktionen, die in Verbindung mit den entsprechenden Variablen aufgerufen werden, müssen für die Klassen bzw. für die Datentypen T, für den das Funktionstemplate verwendet wird, definiert sein.

Breymann_Folien

Funktionstemplates

- Einbinden von Templates:
  - Da die Schablonen zunächst einmal für die entsprechenden Datentypen oder Klassen T durch Textersetzungen in übersetzbare Programme überführt werden, muss man die Template-Dateien mit **#include** einlesen.
  - Beispiel mit Deklaration der Template-Funktionen in der Datei „my_template.h“ und Definition in der Datei „my_template.cpp“:

```
// my_template.h
#ifndef my_template_h
#define my_template_h
// hier folgen die Template-Deklarationen (Prototypen)
//...
//...
//...
// hier werden die Definitionen eingebunden
#include “my_template.cpp“
#endif
```

  - Man beachte: Bei Templates macht es Sinn, die Funktionsdeklarationen und –definitionen in der gleichen Datei zu speichern.

Breymann_Folien

Klassentemplates

- Template Klassen werden auch parametrisierte Datentypen genannt.
- Als Beispiel betrachten wir im folgenden einen parametrisierte Datentyp für unsere einfache Klasse Point2D:

```
// Point2D.t
#ifndef Point2D_t
#define Point2D_t

template<typename T>
class Point2D {
    public:
            T X( ) const { return xCoordinate; }
            T Y( ) const { return xCoordinate; }
            void changeCoordinates( T x , T y) { xCoordinate = x; yCoordinate = y; }
    private:
            T xCoordinate;
            T yCoordinate;
};
#endif
```

- In der Template-Deklaration kann man anstelle von <class T> auch <typename T> schreiben. Diese Schlüsselwort wird heute von den meisten Experten bevorzugt.

Breymann_Folien

Klassentemplates

- Alternative Definition der parametrisierten Methoden der Klasse in der gleichen Datei (außerhalb der Klassendeklaration):

```
// Point2D.t
#ifndef Point2D_t
#define Point2D_t

template<typename T>
class Point2D {
      public:
            T X( ) const;
            T Y( ) const;
            voidchangeCoordinates( T x , T y);
      private:
            T  xCoordinate;
            T  yCoordinate;
};

// Implementierung der Methoden
template<typename T>
T   Point2D<T>::X( ) const { return xCoordinate; }

template<typename T>
T   Point2D<T>::Y( ) const { return yCoordinate; }

template<typename T>
voidPoint2D<T>::changeCoordinates( T x , T y) { xCoordinate = x; yCoordinate = y; }

#endif
```

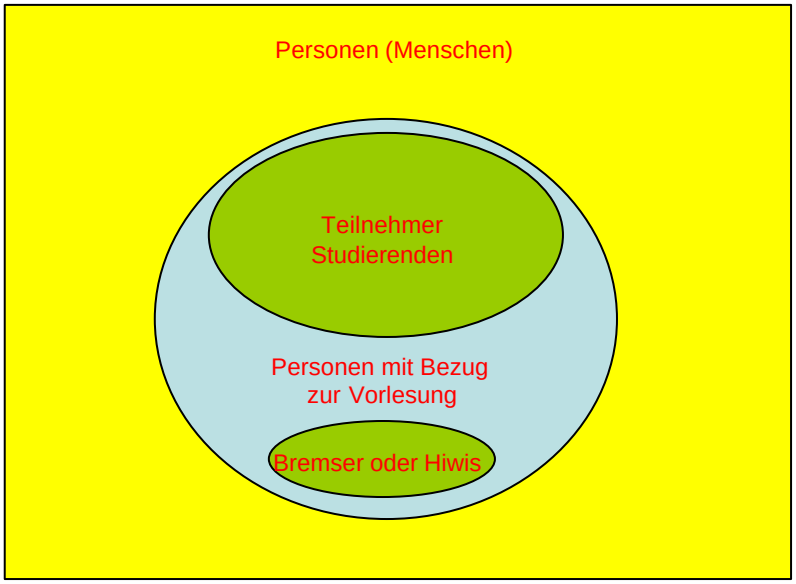
Breymann_Folien

Vererbung

- Das zentrale Ziel des OOP ist
  - Wiederverwendbarkeit von Programmcode
- Das wichtigste Konzept zur Unterstützung der Wiederverwendbarkeit von Programmcode ist die
  - Vererbung
- Was versteht man unter Vererbung?
  - Klassen beschreiben Objekte mit ähnlichen Eigenschaften und ähnlichen Verhaltensweisen (Methoden).
  - Die Objekte einer Klasse können sich dennoch in bestimmten Eigenschaften und Verhaltenweisen unterscheiden.
  - Es kann zum Beispiel Unterklassen geben, die besondere Eigenschaften und Verhaltensweisen besitzen.
  - Für ein Beispiel siehe nächste Seite.

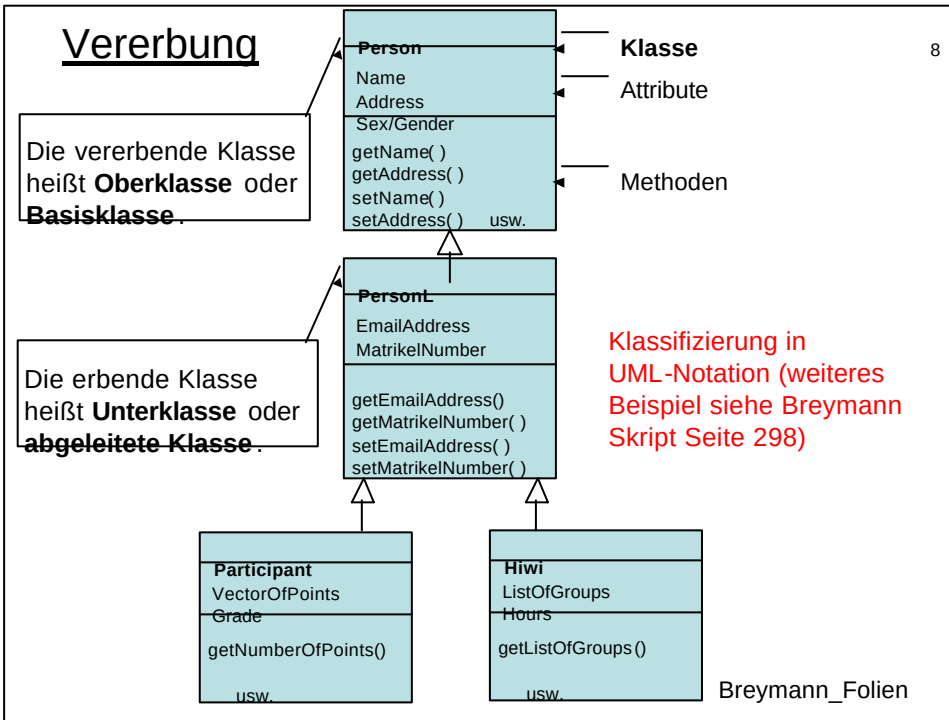
Breymann_Folien

Vererbung



Breymann_Folien

Vererbung



Breymann_Folien

Vererbung und Klassifizierung

- Die Vererbung ist hierarchisch organisiert.
- Sie beschreibt eine so genannte „ist-ein-Beziehung“, d.h.,
  - Ein Objekt der abgeleiteten Klasse ist gleichzeitig auch ein Objekt der Basisklasse.
  - Die Menge der Objekte in der abgeleiteten Klasse sind daher eine Teilmenge der Objekte der Basisklasse.
  - Die abgeleitete Klasse erbt
    - alle Eigenschaften (Attribute) und
    - Verhaltensweisen (Methoden) der Basisklasse.
  - Die Unterklasse ist eine **Spezialisierung** der Oberklasse.
  - Die Oberklasse ist die **Abstraktion** oder **Generalisierung** von ähnlichen Eigenschaften und Verhaltensweisen der Unterklassen.

Vererbung und Klassifizierung

- Die abgeleitete Klasse erbt
  - alle Eigenschaften (Attribute) und
  - Verhaltensweisen (Methoden) der Basisklasse.
- Wenn eine Oberklasse bekannt ist, so brauchen in der abgeleiteten Klasse nur die Abweichungen beschrieben zu werden.
- Ist eine Oberklasse implementiert, so müssen wir für die „neue“ abgeleitete Klasse nur die speziellen Attribute und Methoden der abgeleiteten Klasse implementieren.
- -> Alles andere kann wieder verwendet werden.

Vererbung und Klassifizierung

- Nehmen wir an, dass wir die Basisklasse „Person“ bereits deklariert und implementiert hätten:

```
class Person {
   public:
      void getName( );                      // den Namen ausgeben
      void setName(string pname);           // den Namen setzen
      void getAddress( );                   // die Adresse ausgeben
      void setAddress(string paddress);  // die Adresse
   private:
      string  name;
      string  address;
};
```

- Die Implementierung der Methoden der Klasse „Person“ lassen wir hier außer Acht.

Vererbung und Klassifizierung

- So können wir die Klasse „PersonL(ecture)“ wie folgt von der Klasse „Person“ ableiten:

```
class PersonL : public Person {
   public:
      void getEmailAddress();                    // Emailadresse ausgeben
      void setEmailAddress(string email);        // Emailadresse  setzen
      void getMatrikelNumber( );                    // M.N ausgeben
      void setMatrikelNumber(long matrikelNo);  // M.N. setzen
   private:
      string              emailAddress;
      unsigned long  matrikelNumber;
};
```

- Auf die Diskussion der trivialen Methoden der Klasse „PersonL“ verzichten wir hier.

Vererbung und Klassifizierung

- Jede öffentlich zugängliche Elementfunktion von Person (public) kann auch auf Objekte der Klasse PersonL angewendet werden:

```
PersonL  chefBremser;
chefBremser.setName("Andreas Hildebrandt");              // Person
chefBremser.setEmailAddress("anhi@bioinf.uni-sb.de");    // PersonL
```

- In der ersten Zeile wird der Default-Konstruktor der Klasse PersonL aufgerufen.
- Bevor Speicherplatz für die Attribute der abgeleiteten Klasse PersonL zur Verfügung gestellt wird, wird implizit der Konstruktor der Oberklasse aufgerufen.
- Dieser erzeugt ein anonymes Objekt der Klasse Person und stellt Speicherplatz für diese Objekt zur Verfügung.

Vererbung und Klassifizierung

- Von der Klasse PersonL können wir anschließend die Klasse „Participant" (und natürlich die Klasse Hiwi) ableiten:

```
class Participant: public PersonL {
   public:
      void getNumberofPoints();     // Gesamtpunktezahl berechnen
      void setGrade(float  note);   // Note eingeben
      float getGrade( );            // Note zurückgeben
   private:
      vector<int>      vectorOfPoints;
      float            grade;
};
```

- Diese abgleitete Klasse erbt nun nicht nur von der Klasse PersonL, sondern natürlich auch von der (den) Basisklasse(n) von PersonL (also von Person).

Vererbung und Klassifizierung

15

- Definieren wir in einem Programm ein Objekt der Klasse „Participant“, so können wir alle öffentlichen Element-funktionen der Klasse und der Basisklassen (Person, PersonL) auf dieses Objekt anwenden.

```
Participant  paul;
paul.setName(“Paul Hacker“);                    // Person
paul.setEmailAddress(“paul@rz.uni-sb.de“);   // PersonL

if (paul.getGrade( ) <= 4.0)  cout << “Paul hat bestanden!“ << endl;
```

- Die Konstruktoren werden in der Reihenfolge der Vererbungshierarchie implizit aufgerufen.
- Konstruktoren der Basisklassen können auch explizit in selbst definierten Konstruktoren der abgeleiteten Klasse aufgerufen werden

Breymann_Folien

Vererbung und Klassifizierung

16

- Konstruktoren der Basisklassen können auch explizit in selbst definierten Konstruktoren der abgeleiteten Klasse aufgerufen werden:
  - Entweder im Funktionskörper:

```
Participant::Participant(string pname, string email, long matNo) {
PersonL(email, matNo);   // Hierfür müsste ein entsprechender
                         //  allgemeiner Konstruktor mit diesen
                         //  Parametern definiert werden.
*this.setName(pname);
}
```

  - Oder in der Initialisierungsliste:

```
Participant::Participant(string pname, string email, long matNo)
: PersonL(email, matNo)  {
*this.setName(pname);
}
```

Breymann_Folien

Zugriffsschutz

- Unter Zugriffschutz ist die Abstufung von Zugriffsrechten auf Daten und Elementfunktionen zu verstehen.
- Bisher haben wir zwei Abstufungen kennengelernt:
  - public:
    - Elemente und Methoden unterliegen keiner Zugriffsbeschränkungen.
  - private:
    - Elemente und Methoden sind ausschließlich innerhalb der Klasse zugreifbar, sowie für friend Klassen und –Funktionen.

Zugriffsschutz

- Um abgeleiteten Klassen gegenüber der „Öffentlichkeit“ weitgehende Rechte einräumen zu können, ohne den privaten Status mancher Elemente aufzugeben, gibt es einen weiteren Zugriffsspezifizierer:
  - protected:
    - Elemente und Methoden sind in der eigenen und in allen mit „public“ abgeleiteten Klassen zugreifbar, nicht aber in anderen Klassen oder außerhalb der Klasse.

Zugriffsschutz

- Vererbung von Zugriffsrechten:
  - Gegeben sei eine Oberklasse, von der eine weitere Klasse abgeleitet wird.
  - Für die Vererbung der Zugriffsrechte bei „public“-Vererbung gelten folgende Regeln:

Zugriffsrecht in der Basisklasse	Zugriffsrecht in einer abgeleiteten Klasse
private	Kein Zugriff
protected	protected
public	public

Zugriffsschutz

- Regeln für die Vererbung (public, private, protected):
  - Private Elemente sind in einer abgeleiteten Klasse nicht zugreifbar.
  - In allen anderen Fällen gilt das jeweils restriktivere Zugriffsrecht, bezogen auf die Zugriffsrechte für ein Element und die Zugriffskennung der Vererbung einer Klasse (siehe Beispiel im Breymann-Skript Seite 310-312).

Polymorphismus

- Polymorphismus heißt Vielgestaltigkeit.
- Im OOP ist damit die Fähigkeit gemeint, erst zur Laufzeit eines Programms die zu dem jeweiligen Objekt passende Realisierung einer Operation zu ermitteln:
  - Ein Funktionsaufruf muss irgendwann an eine Folge von auszuführenden Anweisungen gebunden werden.
  - Geschieht es während der Ausführung des Programms, so spricht man vom **dynamischen Binden**.
  - Geschieht es vor der Ausführung des Programms, so spricht man vom **statischen Binden**.
  - Eine zur Laufzeit ausgewählte Methode heißt **virtuelle Funktion**.

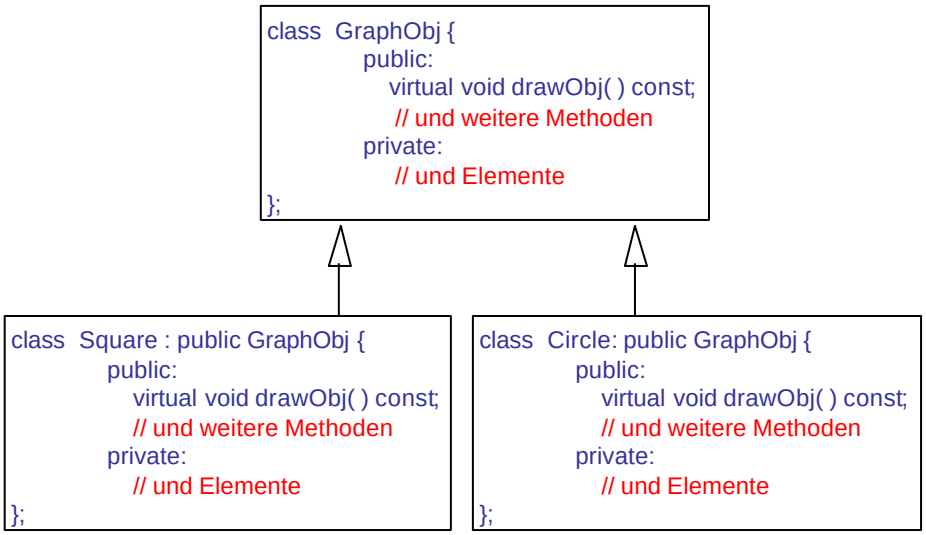
Breymann_Folien

Polymorphismus

- Anhand eines Beispiels aus dem Bereich Computergraphik zeigen wir, welche Vorteile Polymorphismus und dynamisches Binden besitzen.
- Wir betrachten 3 Klassen, eine Basisklasse „GraphObj“ und zwei von „GraphObj“ abgeleitete Klassen „Square“ und „Circle“.
- Diese drei Klassen haben alle eine Elementfunktionen „drawObj“ mit dem gleichem Rückgabetyp „void“ und den gleichen Eingabeparametern.
- Diese Zeichenfunktionen sollen die graphischen Objekte auf dem Bildschirm zeichnen.
- Durch Vorstellen des Schlüsselworts „virtual“ vor die Deklarationen der Funktionen teilen wir dem Compiler mit, dass es sich um virtuelle Funktionen handelt.

Breymann_Folien

Polymorphismus



Breymann_Folien

Polymorphismus

- In einem Graphikprogramm werden eine Reihe von Quadraten und Kreisen in beliebiger Reihenfolge generiert und mittels Zeigern oder Referenzen auf die Basisklasse „GraphObj“ der graphischen Objekte in einem Vektor (Feld) gespeichert.

```
vector<GraphObj*>  vectorOfSquaresAndCircles ;
// oder als
//  vector<GraphObj&>  vectorOfSquaresAndCircles ;
```

- Wir zeichnen nun alle Kreise und Quadrate in einer for-Schleife:

```
for (size_t i = 0; i < vectorOfSquaresAndCircles .size (); i++)
    vectorOfSquaresAndCircles [i]->drawObj();
```

- Erst zur Laufzeit (der for-Schleife) muss die richtige Zeichenfunktion für ein Quadrat oder einen Kreis für jedes im Vektor gespeicherte graphische Objekt dynamisch gebunden werden.
- Da wir virtuelle Elementfunktionen verwendet haben, wird tatsächlich für jeden Kreis und jedes Quadrat die richtige Zeichenfunktion aufgerufen.

Breymann_Folien

Polymorphismus

- Wie funktioniert das dynamische Binden mittels virtueller Funktionen?
  - Die Deklaration einer Funktion als „virtual“ bewirkt, dass Objekten indirekt die Information über den Objekttyp mitgegeben wird und zwar
  - in Form eines Zeigers „vptr“ auf eine besondere Tabelle „vtbl“ (virtual table) mit Zeigern auf die zugehörigen virtuellen Funktionen.
  - Diese Tabelle gehört zu der Klasse des Objekts.
  - Wenn eine virtuelle Funktion über einen Zeiger oder eine Referenz auf dieses Objekt angesprochen wird, weiß das Laufzeitsystem, dass die Funktion über den Zeiger „vptr“ in der Tabelle gesucht und angesprochen werden muss.
  - Hat die Klasse des Objekts keine passende Signatur, so wird eine entsprechende Funktion der Oberklasse gesucht.

Polymorphismus

- Der Einsatz virtueller Funktionen bewirkt, dass Objekten die Typinformation mitgegeben wird.
- Die Objekte werden durch den Zeiger „vptr“ etwas größer.
- Der Umweg über den Zeiger kostet natürlich auch Rechenzeit.
- Man beachte: Dynamisches Binden sollte nur dann verwendet werden, wenn die Klassen (Typen) der Objekte, die angesprochen werden, erst zur Laufzeit bekannt werden (und nicht schon beim Übersetzen feststehen).