Objektorientierte Programmierung
(Winter 2006/2007)

Kapitel 10: Subklassen / Vererbung

- Deklaration von Subklassen
- Vererbung von Daten und Funktionen
- Überschreiben/Redefinition, Virtuelle Funktionen
- Substituierbarkeit, Typ-Umwandlungen
- Mehrfache Vererbung

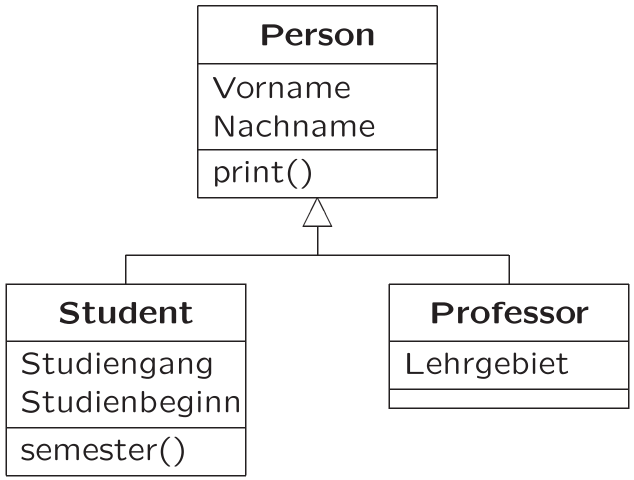
Subklassen (1)

- Oft ist ein Typ von Objekten eine spezielle Art (Spezialfall) eines anderen Typs von Objekten.
- Z.B. könnte man eine Klasse `Person` für Personen an der Universität deklarieren. Diese haben u.a. die Attribute u.a. `Vorname`, `Nachname`.
- Studenten (Klasse `Student`) sind spezielle Personen. Sie haben alle Attribute von `Person`, aber zusätzlich auch Attribute `Studiengang` und `Studienbeginn` (Einschreibe-Datum).

Subklassen (2)

- Professoren (Klasse `Professor`) sind eine andere Art von Personen an der Universität.
- Die Klasse `Person` könnte eine Methode `print()` haben zur Ausgabe von Vorname und Nachname. Diese Methode ist genauso auf Objekte der Klassen `Student` und `Professor` anwendbar.
- Die Klasse `Student` könnte eine Methode `semester()` haben, die das Semester aus dem Studienbeginn berechnet. Diese Methode ist nicht auf Professoren anwendbar (sie haben keinen Studienbeginn).

Subklassen (3)



Subklassen (4)

- `Student` / `Professor` sind Unterklassen / Subklassen der Oberklasse / Superklasse `Person`.
- Stroustrup nennt `Student` (bzw. `Professor`) eine abgeleitete Klasse (engl. “derived class”) der Basisklasse (engl. “base class”) `Person`.
- Zwischen Unterklasse und Oberklasse besteht eine “ist-ein” (engl. “is a”) Beziehung: Ein Student ist eine Person.
- Die Menge der Studenten (Unterklasse) ist eine Teilmenge der Personen (Oberklasse).

Subklassen (5)

- Man nennt diese Beziehung auch eine
  - ◇ Spezialisierung (von `Person` zu `Student`, ein Student ist eine spezielle Person), bzw.
  - ◇ Generalisierung (von `Student` zu `Person`).
- Man betrachtet hier die gleichen Objekte der realen Welt, z.B. die Studentin “Lisa Weiss” auf verschiedenen Abstraktionsebenen:
  - ◇ Mal als Studentin (mit Studiengang etc.),
  - ◇ mal nur als beliebige Person: Hier sind Details wie der Studiengang weggelassen.

Subklassen (6)

- Eine Subklasse “erbt” alle Komponenten (Attribute und Methoden) von der Oberklasse.
- Man braucht für Studenten also nicht explizit zu deklarieren, daß sie einen Vornamen und einen Nachnamen haben, und wie man den vollständigen Namen ausdruckt.
- Man kann Objekte der Unterklasse auch überall verwenden, wo ein Objekt der Oberklasse verlangt wird (Substitutionsprinzip): Die Unterklasse ist ja eine Teilmenge der Oberklasse.

Subklassen (7)

- Z.B. könnte es eine Klasse `Login` für Accounts (Benutzerkonten) auf Rechnern geben.
- Logins werden für alle Personen an der Universität vergeben (Studenten, Professoren, ...).
- Die Klasse `Login` enthält ein Attribut `Inhaber`, das ein Zeiger vom Typ `Person*` ist.
- Dieser Zeiger kann insbesondere auch auf ein Objekt der Klasse `Student` zeigen.

Subklassen (8)

- Eine Unterklasse (abgeleitete Klasse) kann selbst wieder Oberklasse (Basisklasse) für eine andere Unterklasse sein.
- Z.B. ist ein Gastprofessor ein spezieller Professor.

  Als zusätzliche Eigenschaft hat er z.B. das Datum, bis zu dem er an der Uni ist.
- So kann eine ganze Hierarchie von Subklassen entstehen.

Motivation für Subklassen (1)

- Es ist immer gut, mit der Typstruktur des Programms dicht an der realen Welt zu bleiben, und keine künstlichen Verrenkungen machen zu müssen.

  Leichteres Verständnis, weniger Überraschungen.
- Man spart Tippaufwand: `Vorname`, `Nachname`, `print()` hätte man sonst für `Student`, `Professor`, und weitere Arten von Personen einzeln definieren müssen.
- Es wäre auch grundsätzlich schlecht, die gleichen Dinge mehrfach aufschreiben zu müssen.

Motivation für Subklassen (2)

- Falls man etwas für alle Personen ändern möchte (z.B. Namensformat “Nachname, Vorname” statt “Vorname Nachname”), muß man diese Änderung nur an einer Stelle durchführen.
- Ohne Subklassen würde man bei den Logins einen `union`-Typ brauchen, um auf Studenten und Professoren als Inhaber zu verweisen.

  Plus entsprechende Fallunterscheidungen in der Verarbeitung.
- Wenn eine neue Art von Personen hinzukommt, müßte dieser `union`-Typ erweitert werden.

Subklassen: Beispiel (1)

```
typedef const char *str_t;

class Person {
    private:
        str_t Vorname;
        str_t Nachname;
    public:
        str_t get_vorname() const {return Vorname;}
        str_t get_nachname() const {return Nachname;}
        void print()
            { cout << Vorname << ' ' << Nachname; }
        Person(str_t v, str_t n)
            { Vorname = v; Nachname = n; }
};
```

Subklassen: Beispiel (2)

```
class Date {
    private:
        int Day;
        int Month;
        int Year;
    public:
        int get_day()   const {return Day;}
        int get_month() const {return Month;}
        int get_year()  const {return Year;}
        Date(int d, int m, int y)
                { Day = d; Month = m; Year = y; }
};
```


Subklassen: Beispiel (3)

```
class Student : public Person {
    private:
        str_t Studiengang;
        Date Studienbeginn;
    public:
        str_t get_studiengang() const
                {return Studiengang;}
        int semester() const;
        Student(str_t v, str_t n, str_t s,
                             int t, int m, int j) :
            Person(v, n), Studienbeginn(t, m, j)
                { Studiengang = s; }
};
```


Subklassen: Beispiel (4)

- `Student` wird als Subklasse von `Person` deklariert:

  `class Student : public Person { ... };`

- Durch das Schlüsselwort “`public`” sind alle in der Oberklasse als `public` deklarierten Methoden (und ggf. Attribute) auch `public` in der Unterklasse.
- Dies ist der Normalfall, und für das Substitutionsprinzip auch notwendig. Man muß es allerdings explizit hinschreiben, sonst bekommt man `private`.

  Bei `struct` ist dagegen `public` Default (voreingestellt).

Subklassen: Beispiel (5)

- Schreibt man dagegen

  `class Student : private Person { ...};`

  so sind alle Eigenschaften der Oberklasse in der Unterklasse `private` (versteckt) (auch Methoden, die in der Oberklasse `public` waren).
- Dann könnte man auf ein Objekt der Klasse `Student` also z.B. nicht mehr die Methode `get_vorname()` anwenden (außer in Methoden der Klasse `Student`).
- Deswegen gibt es dann z.B. keine automatische Typ-Umwandlung von `Student*` nach `Person*`.

Subklassen: Beispiel (6)

- Man beachte, daß der Konstruktor der Unterklasse den Konstruktor der Oberklasse aufrufen muß (zumindest, wenn es keinen Default-Konstruktor der Oberklasse gibt).

  Die Attribute der Oberklasse müssen ja auch initialisiert werden.

- Dies geschieht im Prinzip in der gleichen Syntax wie Konstruktor-Aufrufe für Komponenten-Objekte.

  Allerdings verwendet man hier den Namen der Oberklasse und nicht den Namen der Komponente. Die Ausführungsreihenfolge ist: Erst die Oberklasse (falls die selbst eine Oberklasse hat: in der Hierarchie von oben nach unten), dann Komponenten-Objekte (in Deklarationsreihenfolge), dann der Rumpf des Konstruktors.

Public/Protected/Private (1)

- Attribute und Methoden, die in der Oberklasse als `private` deklariert sind, sind in der Unterklasse nicht zugreifbar.
- Das ist auch vernünftig, denn sonst könnte der Zugriffsschutz jederzeit durch Deklaration einer Subklasse umgangen werden.

  Der Kreis der Methoden, die auf ein Datenelement zugreifen können, soll ja möglichst klein gehalten werden.

  Soweit ich weiß, kann man in C++ nicht ausschließen, daß noch eine Unterklasse zu einer Klasse deklariert wird. Manche Sprachen und Formalismen haben hierfür ein Schlüsselwort wie “final”.

Public/Protected/Private (2)

- Deklariert man Attribute / Methoden dagegen als “`protected`”, so können Unterklassen darauf zugreifen, nicht aber “fremde” Klassen.
- “`protected`” ist eine Sichtbarkeitsspezifikation wie “`private`” und “`public`”.
- Durch Angaben in der Subklassen-Deklaration, z.B.

  ```
  class Student : protected Person { ...};
  ```

  kann man die Sichtbarkeit von Attributen und Methoden der Oberklasse weiter einschränken (hier `public` → `protected`), aber niemals erweitern.

Implementierung (1)

- Die von der Subklasse neu eingeführten Attribute werden einfach hinten an die Struktur der Oberklasse angehängt.
- Beispiel:
  - ◇ Offset von `Vorname`: 0
  - ◇ Offset von `Nachname`: 4
  - ◇ `sizeof(Person)`: 8
  - ◇ Offset von `Studiengang`: 8
  - ◇ Offset von `Studienbeginn`: 12
  - ◇ `sizeof(Student)`: 24

Implementierung (2)

- Daher braucht der Compiler nichts besonders zu tun, wenn ein `Student*` in ein `Person*` umgewandelt wird:
  - ◇ Die Startadresse des Objekts ändert sich nicht.

    Man greift nur auf den hinteren Teil nicht mehr zu.
- Für Methoden wie `print`, die nur auf die Komponenten `Vorname`, `Nachname` zugreifen, ist egal, ob sie auf ein Objekt der Klasse `Person` oder der Klassen `Student`/`Professor` angewendet werden: Die Offsets der benötigten Komponenten sind gleich.

Überschreiben (1)

- Man kann in einer Subklasse eine Funktion neu definieren, die in der Oberklasse bereits definiert ist.
- In diesem Fall wird die ererbte Funktion überschrieben ("Overriding", "Redefinition").
- Z.B. könnte man in der Subklasse `Professor` die Methode `print()` so definieren, daß sie nach dem Namen "(Professor für ⟨Lehrgebiet⟩)" mit ausgibt.
- Für die Objekte der Subklasse wird jetzt also eine andere Implementierung der Methode `print()` verwendet als für die Objekte der Oberklasse.

Überschreiben (2)

```
class Professor : public Person {
    private:
        str_t Lehrgebiet;
    public:
        str_t get_lehrgebiet() const
            { return Lehrgebiet; }
        void print()
            { Person::print(); // Vorname Nachname
              cout << ' ( Professor für ';
              cout << Lehrgebiet << ')'; }
        Professor(str_t v, str_t n, str_t l)
                : Person(v, n)
            { Lehrgebiet = l; }
};
```


Überschreiben (3)

- Wenn der Compiler eine Methode `f()` für ein Objekt der Klasse `C` aufruft, prüft er zunächst, ob `f()` in dieser Klasse definiert ist. Wenn ja, nimmt er diese Implementierung von `f()`.
- Andernfalls prüft er, ob `f()` in der Oberklasse `B` von `C` definiert ist. Ggf. nimmt er die dort definierte Implementierung.
- Ansonsten steigt er in der Klassenhierarchie sukzessive weiter auf, notfalls meldet er die Methode am Ende als undefiniert.

Überschreiben (4)

- C++ erlaubt Funktionen mit dem gleichen Namen, aber verschiedenen Argumenttypen (→ Kapitel 12).
- Eine Funktion wird in der Subklasse nur überschrieben, wenn man dort eine Funktion definiert mit
  - ◇ dem gleichen Namen,
  - ◇ der gleichen Parameter-Anzahl und den gleichen Parameter-Typen, und
  - ◇ dem gleichen Resultat-Typ.

    Der Resultat-Typ könnte tatsächlich seine Subklasse des entsprechenden Typs der ererbten Funktion sein.
- Sonst: Zwei Funktionen in der Subklasse.

Überschreiben (5)

- Nun gibt es allerdings ein Problem:
  - ◇ Wenn man einen Zeiger auf ein `Professor`-Objekt in einen Zeiger auf ein `Person`-Objekt umwandelt, und dann dafür die Methode `print()` aufruft, weiß der Compiler nicht mehr, daß es eigentlich ein `Professor`-Objekt war.
  - ◇ Daher wird die bei `Person` definierte Implementierung aufgerufen (die den Namen ohne Zusatz ausgibt).
  - ◇ Manchmal ist das, was man will.

Überschreiben (6)

- Um immer die für die ursprüngliche/tatsächliche Klasse des Objektes definierte Funktion aufzurufen, selbst nach einer Typumwandlung in eine Oberklasse, muß man die Funktion als "`virtual`" deklarieren:

```
class Person {
    public:
        virtual void print();
        ...
};
class Professor : public Person {
    public:
        void print() { ... }
        ...
};
```


Überschreiben (7)

- Der Compiler könnte dies z.B. so implementieren, daß er die Adresse der tatsächlich aufzurufenden Funktion in dem Objekt selbst speichert (gewissermaßen in einem versteckten Attribut).
- Häufig haben solche Klassen aber viele virtuelle Funktionen. Dann würden die Objekte durch die vielen Funktionszeiger zu sehr aufgebläht werden.
- Daher ist üblich, nur einen Zeiger auf eine "Virtual Function Table" zu speichern (siehe nächste Folie).

Überschreiben (8)

- Die Virtual Function Table enthält die Startadressen aller virtuellen Funktionen der Klasse (inklusive der geerbten).
- Diese Tabelle legt der Compiler einmal für jede Klasse mit virtuellen Funktionen an.
- Der Zeiger auf die "Virtual Function Table" im Objekt entspricht einer Typ-Identifikation.

  Da der Compiler die exakten Typen der Objekte zur Compilezeit nicht kennt, muß er Code erzeugen, der zur Laufzeit die richtige Implementierung auswählt.
- Der Funktionsaufruf ist so etwas langsamer.

Überschreiben (9)

- Wenn man das (mit MS Visual C++ 6.0) ausprobiert, stellt man folgendes fest:
  - ◇ Offset von `Vorname`: 4
  - ◇ Offset von `Nachname`: 8
  - ◇ `sizeof(Person)`: 12
  - ◇ Offset von `Studiengang`: 12
  - ◇ Offset von `Studienbeginn`: 16
  - ◇ `sizeof(Student)`: 28
- Auch wenn man weitere virtuelle Funktionen deklariert, ändert sich daran nichts.

Überschreiben (10)

- Wenn man unter dem gleichen Namen verschiedene Implementierungen der Funktion aufruft, heißt die Funktion polymorph ("vielgestaltig").
  - ◇ Virtuelle Funktionen in C++ werden daher auch polymorphe Funktionen genannt.
  - ◇ Entsprechend heißt eine Klasse mit polymorphen Funktionen eine polymorphe Klasse.
- Konstruktoren können nie virtuell sein (mindestens dann ist die genaue Klasse ja bekannt).

Überschreiben (11)

- Falls man den Speicher für Objekte einer Subklasse dynamisch anfordert (→ Kapitel 11), darf man sie über Zeiger auf die Oberklasse nur dann löschen, wenn der Destruktor `virtual` ist.

  Z.B. sind `Student`-Objekte ja größer als `Person`-Objekte. Man kann ein `Student`-Objekt aber wie ein `Person`-Objekt verwenden. Wenn man dem Compiler nun aber mitteilt, daß man ein `Person`-Objekt löschen will, und es handelt sich tatsächlich um ein `Student`-Objekt, wird der `Student`-spezifische Teil des Speichers nicht freigegeben — es sei denn, der Destruktor ist `virtual` (dann ist ja zur Laufzeit bekannt, wie groß das Objekt wirklich ist).

Überschreiben (12)

- Eine Funktion muß in der obersten Klasse, in der sie eingeführt wird, als `virtual` deklariert werden.
- Man kann sie nicht nachträglich in einer Unterklasse `virtual` machen.

  Das Schlüsselwort `virtual` wird in der Unterklasse nicht wiederholt.
- Entsprechend sind Unterklassen einer polymorphen Klasse automatisch wieder polymorph.

Typ-Umwandlungen (1)

- Wie oben erläutert, kann man immer und überall ein Objekt der Unterklasse als ein Objekt der Oberklasse auffassen (bei `public`-Vererbung).

```
void f(Person p);
void g(Person& p);
void h(Person* p);
void test()
{       Student s;
        Person x = s;
        Person& y = s;
        Person* z = &s;
        f(s); g(s); h(&s);
}
```


Typ-Umwandlungen (2)

- Eine (explizite) Typumwandlung heißt engl. "cast".

  Rollenbesetzung beim Film, auch Gipsverband (Typ-Umwandlungen sind eher verpönt, letzte Rettung bei etwas brüchigem Algorithmus).
- Eine Typumwandlung von der Unterklasse in die Oberklasse ("Up-Cast") geschieht implizit.
- Bei Zuweisungen oder bei der Parameterübergabe "call by value" werden nur die Attribute kopiert, die die Oberklasse auch hat, d.h. ein Teil des Objektes wird abgeschnitten ("slicing").

Typ-Umwandlungen (3)

- Wenn man mit Zeigern oder Referenzen arbeitet, zeigen die aber noch auf das vollständige Objekt der Unterklasse.
- Ein Zugriff auf Methoden oder Attribute der Unterklasse über einen Zeiger auf die Oberklasse würde zu einem Typfehler führen.
- Z.B. kann man für `Person* p` nicht direkt

  `p->semester()`

  aufrufen, selbst wenn man `p` vorher die Adresse von einem `Student`-Objekt zugewiesen hat.

Typ-Umwandlungen (4)

- Man kann aber auf eigene Gefahr einen "Down-Cast" anwenden:

  `static_cast<Student*>(p)->semester()`

- Bei polymorphen Klassen ist folgendes sicherer:

  `Student* x = dynamic_cast<Student*>(p);`

- Über den Zeiger auf die "Virtual Function Table", den das Objekt enthält, kann geprüft werden, ob es tatsächlich ein `Student` ist. Dann findet die Typ-Umwandlung statt. Ansonsten wird `x` auf `0` gesetzt.

Abstrakte Klassen (1)

- Eine abstrakte Klasse ist eine Klasse, die selbst keine direkten Instanzen hat.
- Wenn z.B. `Person` eine abstrakte Klasse wäre, so wären alle Objekte dieser Klasse eigentlich Objekte der Unterklassen `Student` oder `Professor`.

  Objekte der Unterklasse gehören ja immer automatisch mit zur Oberklasse, zumindest findet bei Bedarf eine implizite Typ-Umwandlung statt.

- Man sagt in diesem Fall, daß die Spezialisierung total ist: Jedes Objekt der Oberklasse gehört zu einer Unterklasse.

Abstrakte Klassen (2)

- Abstrakte Klassen sind nützlich als gemeinsames Interface für ihre Unterklassen.
- Manchmal möchte man in der abstrakten Klasse eine Methode deklarieren (so daß sie Bestandteil der Schnittstelle wird), kann sie aber noch nicht sinnvoll definieren (implementieren).
- Wenn man z.B. eine abstrakte Klasse für geometrische Objekte in der Ebene definiert (Kreise, Rechtecke, etc.), so würde es Sinn machen, eine Methode zur Berechnung des Flächeninhaltes vorzusehen.

Abstrakte Klassen (3)

- Man kann die Flächenberechnung aber für allgemeine geometrische Objekte nicht sinnvoll implementieren (das geht nur für die Subklassen).
- Die Lösung sind "pure virtual functions":

  `virtual float area() = 0;`

  Da diese Funktion das Objekt nicht ändert, geht auch: `virtual float area() const = 0;`

- Ohne "`= 0`" würde man beim Linken einen Fehler bekommen, wenn die Virtual Function Table zusammengesetzt wird.

Abstrakte Klassen (4)

- Wenn man eine "pure virtual function" deklariert, weiß der Compiler, daß es sich um eine abstrakte Klasse handelt.

  Wenn man niemals Objekte einer Oberklasse erzeugt, wäre es theoretisch auch eine abstrakte Klasse, aber der Compiler unterstützt den Programmierer dann natürlich nicht bei der Einhaltung/Überwachung dieser Bedingung.

- Man kann dann keine Objekte der Klasse erzeugen.
- Man kann aber natürlich Objekte einer Unterklasse erzeugen, in der die "pure virtual functions" mit konkreten Funktionen überschrieben sind.

Mehrfache Vererbung (1)

- Manchmal hat man mehrere orthogonale Klassifizierungen, und Objekte müssen zu mehreren, von einander unabhängigen Klassen gehören.
- Z.B. ist ein Tutor (wissenschaftliche Hilfskraft)
  - ◇ ein Student und
  - ◇ ein Angestellter der Universität.
- Als Student hat er eine Matrikelnummer, einen Studiengang, etc.
- Als Angestellter hat er eine Personalnummer, eine Bankverbindung, etc.

Mehrfache Vererbung (2)

- C++ erlaubt mehrfache Vererbung, d.h. eine Klasse kann gleichzeitig mehrere Oberklassen haben.
- Es ist also folgende Deklaration möglich:

```
class Tutor : public Student, public Personal {
          string Betreuer;
          ...
}
```

- Ein Objekt der Klasse `Tutor` gehört also gleichzeitig auch zu den Klassen `Student` und `Personal`.
- Es erbt alle Attribute und Methoden von `Student` und von `Personal`, und fügt dann noch eigene hinzu.

Mehrfache Vererbung (3)

- Mehrfache Vererbung kann zu Mehrdeutigkeiten bei Attributen und Funktionen führen:
  - ◇ `Student` und `Personal` können Komponenten haben, die zufällig gleich benannt wurden.

    Es ist ein anderes Problem, wenn dies tatsächlich nicht zufällig ist, sondern von einer gemeinsamen Oberklasse wie `Person` herrührt. Dieser Fall wird unten noch betrachtet.

  - ◇ Beispiel: `Student` hat ein Attribut `Nr` (Matrikelnummer), `Personal` auch (Personalnummer).
  - ◇ Natürlich gibt es in `Tutor` dann beide Attribute.

Mehrfache Vererbung (4)

- Mehrdeutige Attribut-/Funktionsnamen, Forts.:
  - ◇ Man kann das Attribut/die Funktion für ein Objekt t der Klasse `Tutor` dann unter Angabe eines Klassennamens eindeutig machen:

```
cout << t.Student::get_nr();
```

Wenn die Hierarchie größer ist, braucht man nicht unbedingt die Klasse zu verwenden, in die jeweilige Komponente definiert wurde, es reicht eine Klasse, von der aus es nach oben eindeutig ist.

Man könnte `get_nr` in `Tutor` überschreiben, und dort z.B. die Matrikelnummer liefern, und eine weitere Methode `get_persnr` für die Personalnummer einführen.

Man kann die Situation auch eindeutig machen, indem man in der Klassendeklaration von `Tutor` schreibt: "`using Student::get_nr();`".

Mehrfache Vererbung (5)

- Wie sonst auch, kann ein Objekt einer Subklasse jederzeit wie ein Objekt einer Oberklasse verwendet werden, auch über Zeiger und Referenzen.
- Nun ergibt sich ein Implementierungsproblem:
  - ◇ Bei einfacher Vererbung können die Attribute der Subklasse einfach hinten an die Attribute der Oberklasse angehängt werden.
  - ◇ Die Attribute der Oberklasse haben also einen festen Offset vom Beginn des Objektes, egal ob es zur Unterklasse oder zur Oberklasse gehört.

Mehrfache Vererbung (6)

- Das geht bei mehrfacher Vererbung nicht mehr:
  - ◇ In den Oberklassen sind die Adressen der Attribute unabhängig von einander belegt worden (jeweils von 0 beginnend).
  - ◇ In der Unterklasse müssen die Attribute der beiden Oberklassen hintereinander abgelegt werden.
  - ◇ Bei der Typumwandlung von `Tutor*` in `Personal*` muß auf den Zeiger jetzt die Größe von `Student` addiert werden (um die Basisadresse des Personalanteils zu bekommen).

Mehrfache Vererbung (7)

- Typumwandlung bei mehrfacher Vererbung, Forts.:
  - ◇ Während die Typumwandlung bei Zeigertypen sonst nur dazu dient, Fehlermeldungen des Compilers abzuschalten, tut sie hier wirklich etwas (ändert den Wert).

    Eine Typumwandlung zwischen `int` und `float` würde auch das Bitmuster grundlegend ändern. Es ist also gar nicht so neu.
  - ◇ Umgekehrt muß auch bei der Typumwandlung zurück von `Personal*` nach `Tutor*` die Größe von `Student` wieder abgezogen werden.

Mehrfache Vererbung (8)

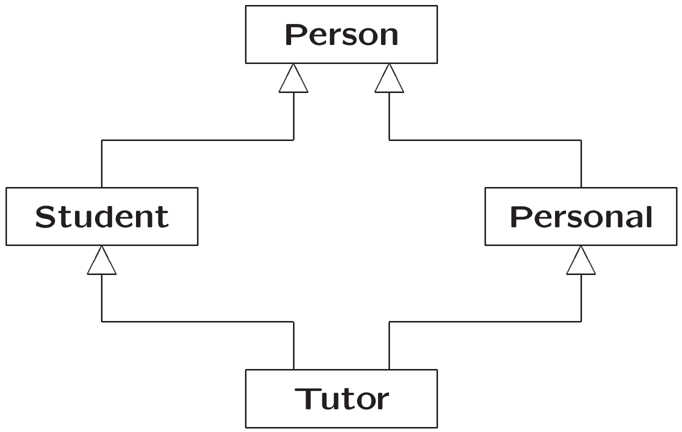
- Typumwandlung bei mehrfacher Vererbung, Forts.:
  - ◇ Entsprechend muß beim Aufruf einer in `Personal` deklarierten Methode für ein `Tutor`-Objekt nicht die Startadresse des Objektes als “`this`” übergeben werden, sondern die des `Personal`-Anteils.
  - ◇ Die “Virtual Function Table” braucht jetzt jeweils noch ein Feld für die Adressdifferenz.

    Z.B. könnte ein Zeiger vom Typ `Personal*` gegeben sein, über den eine Funktion aufgerufen werden soll. In Wirklichkeit zeigt der Zeiger aber auf den `Personal`-Anteil eines `Tutor`-Objektes. Wenn die Funktion in `Tutor` überschrieben ist, muß beim Funktionsaufruf die Größe von `Student` vom `this`-Zeiger abgezogen werden.

Mehrfache Vererbung (9)

- Man kann nicht direkt zweimal die gleiche Oberklasse angeben, aber es ist möglich, daß die Oberklassen einer Klasse irgendwo in der Hierarchie eine gemeinsame Oberklasse haben.
- Z.B. sind `Student` als auch `Personal` von der gemeinsamen Oberklasse `Person` abgeleitet.
- Normalerweise bedeutet das in C++, daß ein `Tutor`-Objekt zwei verschiedene `Person`-Objekte enthält: Einmal als Teil des `Student`-Objektes, und einmal als Teil des `Personal`-Objektes.

Mehrfache Vererbung (10)



Mehrfache Vererbung (11)

- Im Beispiel ist das nicht beabsichtigt, z.B. würden Vorname und Nachname zweimal abgespeichert.

  Lösung siehe nächste Folie.
- Falls es aber beabsichtigt wäre, müßte man die Mehrdeutigkeiten berücksichtigen:
  - ◇ “`t.Person::get_vorname()`” wäre nicht eindeutig, man muß “`t.Student::get_vorname()`” schreiben.
  - ◇ Eine Typumwandlung von `Tutor*` in `Person*` wäre nicht eindeutig, man muß z.B. erst in `Student*` und dann weiter in `Person*` umwandeln.

Mehrfache Vererbung (12)

- Falls man (wie im Beispiel) nur eine Ausprägung der gemeinsamen Oberklasse wünscht, muß man die gemeinsame Oberklasse als `virtual` deklarieren:

```
class Person { ... };
class Student : public virtual Person { ... };
class Personal : public virtual Person { ... };
class Tutor : public Student, public Personal
          { ... };
```


Mehrfache Vererbung (13)

- In diesem Fall enthalten `Student` und `Personal` implizit einen Zeiger auf ein `Person`-Objekt, der bei der Typumwandlung oder beim Zugriff auf Attribute aus `Person` dereferenziert werden muß.

  In einen `Student`-Objekt würde der `Person`-Teil z.B. direkt hinter den `Student`-Attributen abgelegt (umgekehrt wie bei normaler Vererbung). Wenn das `Student`-Objekt aber Teil eines `Tutor`-Objektes ist, stehen dazwischen noch die Daten von `Personal`. Weil es also keinen festen Offset gibt, braucht man den Zeiger.

- Typumwandlungen von `Student*` nach `Person*` sind möglich (der implizite Zeiger wird dereferenziert), in der umgekehrten Richtung jetzt nicht mehr.

Mehrfache Vererbung (14)

- Häufig kann man den (nicht unterstützten) Down-Cast mit virtuellen Funktionen ersetzen.

  Wenn die Virtual Function Table im `Person`-Teil eines `Tutor`-Objektes angelegt wird, kann für Funktionen, die in `Tutor` überschrieben sind, die Adress-Differenz zum `Tutor`-Objekt eingetragen werden.

- Bei virtuellen Basisklassen muß man aufpassen, daß Funktionen nicht doppelt aufgerufen werden.

  Z.B. könnten `Student` und `Personal` eine Funktion für ihr "Oberobjekt" aufrufen, was einzeln auch funktionieren würde, aber wenn `Tutor` die Funktionen von `Student` und `Personal` aufruft, wird die Funktion auf dem geteilten Oberobjekt zweimal ausgeführt. Für den Konstruktor gibt es deswegen die Regel, das er vom untersten Objekt aufgerufen wird, d.h. der Konstruktor von `Tutor` ruft den von `Person` auf.

Mehrfache Vererbung (15)

- Aufgrund all dieser Probleme wird mehrfache Vererbung nur in einigen objektorientierten Sprachen angeboten, neben C++ z.B. auch Eiffel, Python.
- Sprachen wie Java, C# bieten zwar nur Einfach-Vererbung, aber eine Klasse kann dort beliebig viele "Schnittstellen" ("Interfaces") implementieren:
  - ◇ Die Klasse garantiert, daß sie bestimmte Funktionen anbietet, aber erbt keine Attribute oder Implementierungen.

    Dieses Konzept ist sehr ähnlich zu einer abstrakten Oberklasse mit ausschließlich "pure virtual functions".