UMIT
the health & life sciences university

Vorlesung 3P: Ableitung und Vererbung

- Move-Semantik und Rvalue-Referenzen
- Ableitung und Vererbung

Temporäre Objekte

- Wird von einer Methode/Funktion ein *neues* Objekt erzeugt und zurückgegeben, so ist dies allein mit Copy-Konstruktoren und kopierenden =-Überladungen in manchen Situationen ineffizient.

  Beispiel (mit überladenen Operatoren + und =):

```
class Test { ... };
...
Test a, b, c;
...
c = a + b;
```

  Hier muss `a + b` ein neues Objekt erzeugen, das dann durch = in `c` kopiert und anschließend sofort vernichtet wird (ein *temporäres* Objekt).

```
Test Test::operator+ (const Test &x) {
  Test result;
  ...
  return result;
}
```

  Zwar optimiert ein moderner Compiler die Kopie von `result` in den Rückgabewert weg, aber der Kopiervorgang bei = bleibt.

Move-Semantik und Rvalue-Referenzen

- Seit C++11 ist es möglich, den =-Operator so zu definieren, dass er eine *Rvalue-Referenz* als Argument bekommt und das Objekt in `c` verschiebt, statt es zu kopieren:

  ```
  Test& Test::operator= (Test&& source) {...}
  ```

  Mit `&&` wird die Rvalue-Referenz gekennzeichnet, der Operator `=` hat *Move-Semantik*.

- Faustregel: Eine Lvalue-Referenz gehört zu einem Objekt, das mit einem Namen angesprochen werden kann, eine Rvalue-Referenz zu einem Objekt, das keinen Namen hat.

UMIT
the health & life sciences university

Beispiel zur Nacharbeit: Quellcode verstehen

→2301-images-class

- Legen Sie ein neues C++-Konsolenprojekt mit den Quellcode- und Header-Dateien aus dem (in Moodle bereit gestellten) Ordner `2301-images-class-src` an
- Finden Sie die Änderungen gegenüber dem vorangehenden Beispiel und versuchen Sie diese zu verstehen

  Aus `image.h`*:*

```
double upixel (const int i, const int j) const;
double& upixel (const int i, const int j);
double pixel(int i, int j) const;
double& pixel(int i, int j);
double operator() (const int i, const int j) const;
double& operator() (const int i, const int j);
Image (const Image&);
Image (Image&&);
Image& operator= (const Image&);
Image& operator= (Image&&);
```

Abgeleitete Klassen

- Abgeleitete Klassen dienen dazu, die Funktionalität einer Klasse zu *erweitern*.
- Von einer *Basisklasse,* deren grundlegende Eigenschaften einmal festgelegt wurden, können weitere Klassen abgeleitet werden, die zusätzliche Eigenschaften (Datenelemente) besitzen und zusätzliche Fähigkeiten (Methoden) bieten.
- Dadurch wird eine bessere Abstraktion erreicht: Gemeinsame Aspekte ähnlicher Objekte werden in einer Basisklasse zusammengefasst.
- *Beispiel:* Digitalfotos, Computertomografie-Bilder, MRI-Bilder haben alle die grundlegenden Eigenschaften und Fähigkeiten eines Bildes; hinzu treten aber jeweils zusätzliche Besonderheiten wie etwa spezifische Metadaten (für Fotos: Exif-Datenfelder, evtl. GPS-Daten; für medizinische Bilder Aufnahmeparameter, Patient/inn/endaten)
- *Weiteres Beispiel:* File-Streams bieten die Möglichkeiten von Streams (Ein- und Ausgabe), aber zusätzlich die Fähigkeit, Dateien zu öffnen und zu schließen, die Schreib-/Leseposition zu verschieben usw.

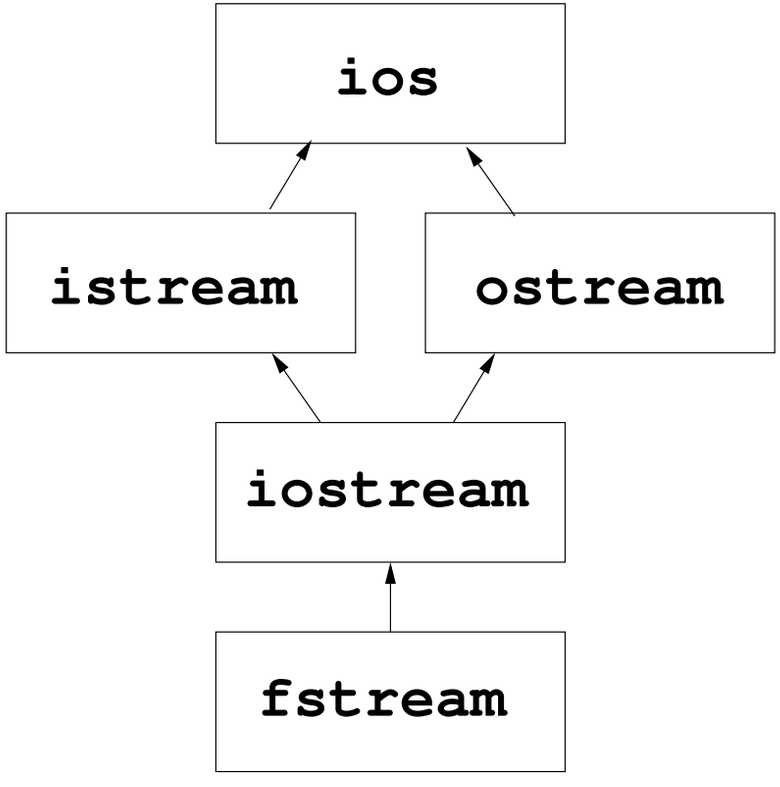
Vererbung

- Ein wesentlicher Aspekt der OOP ist, dass Objektklassen voneinander abgeleitet werden können und damit *Hierarchien* bilden.
- **Vererbung** bezeichnet dabei die Tatsache, dass Objekte abgeleiteter Klassen die Eigenschaften der Basisklasse (von der sie abgeleitet werden) übernehmen.

  Eigenschaften umfassen dabei sowohl *Datenelemente* als auch *Methoden.*
- Abgeleitete Klassen können zusätzliche Eigenschaften besitzen, die Eigenschaften der Basisklasse ergänzen und auch mitunter ersetzen können.
- *Beispiel:* `ofstream`
  - Ein `ofstream` ist ein `ostream` mit zusätzlichen Fähigkeiten (für die Dateiarbeit). Die Fähigkeiten zur Ausgabe in den Stream werden von `ostream` an `ofstream` vererbt.

  Beispiel: `fstream`
  - Ein `fstream`, also ein Filestream mit Ein- und Ausgabefähigkeiten, ist von `iostream` abgeleitet, der seinerseits sowohl von `istream` als auch von `ostream` abgeleitet ist (*Mehrfachvererbung*).

Vererbung



Darstellung der Klassenhierarchie zur Ableitung der Klasse `fstream` als Klassendiagramm. Pfeile zeigen von den abgeleiteten zu den Basisklassen.

Beispiel zur Ableitung

- *Beispiel:* Ein *Bild mit Metainformationen* ist ein Bild, das zusätzliche Eigenschaften und Fähigkeiten hat (Metainformationen lesen, ausgeben, verändern).
- *Syntax:*

```
class ImageMeta : public Image {
  ...
}
```

- Die Datenelemente und Methoden der Basisklasse sind in der abgeleiteten Klasse enthalten (ohne neue Deklaration) – sie werden **vererbt.**

  Beispiel: `nx`, `ny`, Pixelfeld `p` (Datenelemente), `sizex()`, `sizey()`, Pixelzugriff (Methoden)

- Zu den Datenelementen und Methoden der Basisklasse können weitere Elemente hinzudefiniert werden

  Beispiel: Kommentar (Datenelement), Operationen zum Lesen, Schreiben, Verändern des Kommentars (Methoden)

- Dabei können Methoden der Basisklasse ersetzt (überschrieben) werden – z. B. muss die `readpgm`-Methode durch eine erweiterte Version ersetzt werden, die auch die Kommentare aus dem PGM-File einliest.

UMIT
the health & life sciences university

Beispiel: Quellcode verstehen

→2302-images-class

- Wir betrachten, wie im Beispiel 2302-images-class-src die neu hinzugekommenen Features (Verarbeitung der Kommentarzeilen aus dem PGM-Header) umgesetzt sind

Aus imagemeta.h *:*

```
class ImageMeta : public Image {
  private:
    vector <string> comment; // holds comment lines
    bool readpgm (const string filename);
  public:
    ImageMeta (const int nx, const int ny,
               vector <string> comment = {} );
    ImageMeta (const string filename);
    bool writepgm (const string filename) const;
    vector <string> & accesscomment ();
};
```

Abgeleitete Klassen im Vergleich mit Elementklassen

- Wir haben bereits gesehen, dass Klassen andere Klassen als Datenelemente enthalten können. Wozu braucht man noch eine weitere Möglichkeit, Klassen zu „verschachteln“?
- Die Elemente einer Elementklasse sind „eine Verschachtelungsebene tiefer“, man müsste also zwei Punktoperatoren benutzen, um auf ein Element zuzugreifen.

  Beispiel: Würde die `Image`-Funktionalität in der Klasse `ImageMeta` durch ein *Datenelement* wie `Image theimage` eingebettet, so müsste die Bildgröße mit `myimagemeta.theimage.sizex()` statt einfach mit `myimagemeta.sizex()` abgefragt werden.

  Analog müsste ein Pixel mit `myimagemeta.theimage (i, j)` abgefragt werden statt einfach mit `myimagemeta (i, j)`.

  Allgemein würden Methoden der eingebetteten Klasse nicht zu Methoden der neuen Klasse.
- Eigentlich möchte man aber überall, wo es auf die zusätzliche Funktionalität des `ImageMeta` nicht ankommt, dieses wie ein gewöhnliches Bild behandeln können!

  Genauso bei Filestreams: Immer wenn es nur auf Ein-/Ausgabe ankommt, möchte man diese nicht anders behandeln als z. B. `cin` *und* `cout`*.*

Wann ableiten, wann Elementklassen?

- Ableitung von Klassen und Elementklassen haben unterschiedliche Rollen in Objektmodellen!
- Mit einer Elementklasse wird eine **HAT-Beziehung („has a“)** ausgedrückt, z. B.
  - Ein Kalenderdatum *hat* einen Tag, Monat und Jahr (vgl. Vorlesung 9).
  - Ein Bild *hat* ein Pixelfeld.
  - Ein Verkehrsmittel *hat* z. B. einen Antrieb und Plätze für Passagiere.
  - Ein Kraftfahrzeug *hat* Räder, aber auch einen Antrieb und Plätze.
  - Ein Pkw *hat* Antrieb, Plätze, Räder usw.
- Eine abgeleitete Klasse drückt eine **IST-Beziehung („is a“)** aus, z. B.
  - Ein Bild mit Metainformationen *ist* ein Bild.
  - Ein Pkw *ist* ein Kraftfahrzeug, und ein Kraftfahrzeug *ist* ein Verkehrsmittel.
- Das Kraftfahrzeug „erbt“ Antrieb und Plätze vom Verkehrsmittel, der Pkw erbt beides und die Räder vom Kraftfahrzeug.

UMIT
the health & life sciences university

Beispiel: Quellcode verstehen

→2302-images-class

- Wir betrachten noch einmal das Projekt `2302-images-class` und darin die Implementation des Konstruktors `ImageMeta (const int nx, const int ny, vector <string> comment )`. Was fällt Ihnen daran auf?

```
ImageMeta::ImageMeta (const int nx, const int ny,
                      vector <string> comment )
  : Image (nx, ny), comment (comment)
{}
```

Basisinitialisierer

- In

```
ImageMeta::ImageMeta (const int nx, const int ny,
                      vector <string> comment )
  : Image (nx, ny), comment (comment)
```

  ist `comment (comment)` ein Elementinitialisierer, wie wir ihn bereits kennen; er initialisiert den `vector<string> comment` etwas effizienter, als wenn das erst im Funktionskörper des Konstruktors gemacht würde.

- Der Eintrag `Image (nx, ny)` initialisiert in ähnlicher Weise das eingebettete Basisklassenobjekt.

  Achtung: Während im Elementinitialisierer das Datenelement (Instanz) steht, steht im Basisinitialisierer der Klassenname! (Daran kann man Element- und Basisinitialisierer auch sicher unterscheiden.)

- Auch der Basisklasseninitialisierer ermöglicht eine effizientere Initialisierung.

- Darüber hinaus wäre hier die nachträgliche Zuweisung von `nx`, `ny` und Allokation von Speicher in `p` im Konstruktor von `ImageMeta` gar nicht möglich, da diese Datenelemente privat in `Image`, also auch für die abgeleitete Klasse unzugänglich sind.

Konstruktoren, Chaining und Basisinitialisierer

- Wie bei Elementklassen erfolgt auch bei abgeleiteten Klassen ein *Chaining* der Konstruktoren und Destruktoren:

- Wird eine abgeleitete Klasse instanziiert, so wird zuerst der Konstruktor der Basisklasse aufgerufen, danach der der abgeleiteten Klasse.

  Beispiel: Wird eine Instanz von `ImageMeta` erzeugt, so wird erst der Konstruktor `Image::Image` abgearbeitet, dann `ImageMeta::ImageMeta`.

- Ohne Basisinitialisierer wird immer der Default-Konstruktor der Basisklasse aufgerufen!

  Basisinitialisierer bewirken, dass ein passender anderer Konstruktor aufgerufen wird.

  In den anderen Konstruktoren von `ImageMeta` (außer `ImageMeta (nx, ny, comment)`) wird momentan die Basisklasse noch mit ihrem Default-Konstruktor initialisiert.

- Für Destruktoren gilt die umgekehrte Aufrufreihenfolge.

Zugriffstypen und Ableitung

- Meist wird eine Klasse mit

  ```
  class NeueKlasse : public Basisklasse { ... }
  ```

  abgeleitet. Dann gilt:

  - Die `public`-Elemente der Basisklasse sind auch öffentliche Elemente der abgeleiteten Klasse.
  - Die `private`-Elemente der Basisklasse bleiben privat. Sie sind nur für Methoden der Basisklasse zugänglich (und auch für Methoden der abgeleiteten Klasse unzugänglich).

- Neben `public` und `private` gibt es als dritte Möglichkeit `protected`.

  - Als `protected` deklarierte Elemente der Basisklasse sind für Methoden der Basisklasse und der abgeleiteten Klasse zugänglich, jedoch vor jedem weiteren Zugriff geschützt.
  - Bezüglich weiterer Ableitungen sind `protected`-Elemente der Basisklasse auch `protected` in der abgeleiteten Klasse.

UMIT
the health & life sciences university

Beispiel: Quellcode verstehen

→2302-images-class

- Wir betrachten noch einmal das Projekt 2302-images-class.

  In main.cpp ist der Programmteil der main-Funktion zum Rotieren von Bildern auskommentiert. Was passiert, wenn wir diese Zeilen wieder hinzunehmen?

```
$ g++ --std=c++11 -o 1007-images-class *.cpp

main.cpp: In function ‘int main(int, char**)’:
main.cpp:68:13: error: no match for ‘operator=’ (operand types are ‘ImageMeta’
and ‘Image’)
     myimage = rotateimage (myimage);
             ^
In file included from main.cpp:4:0:
imagemeta.h:9:7: note: candidate: ImageMeta& ImageMeta::operator=(const
ImageMeta&)
 class ImageMeta : public Image {
       ^
imagemeta.h:9:7: note:   no known conversion for argument 1 from ‘Image’ to
‘const ImageMeta&’
imagemeta.h:9:7: note: candidate: ImageMeta& ImageMeta::operator=(ImageMeta&&)
imagemeta.h:9:7: note:   no known conversion for argument 1 from ‘Image’ to
‘ImageMeta&&’
```

UMIT
the health & life sciences university

Beispiel: Quellcode verstehen

→2303-images-class

- Wir gehen jetzt zum Folgebeispiel 2303-images-class. Hier funktioniert das Rotieren von Bildern in main wieder.

  In imagemeta.h *:*

  ```
  ImageMeta (const Image&);
  ImageMeta& operator= (const Image&);
  ```

- *Nacharbeit:* Achten Sie außerdem auf die gegenüber 2302-images-class eingefügten zusätzlichen Basisinitialisierer bei allen Konstruktoren von ImageMeta.

Zuweisungen und Copy-Konstruktoren

- Die Zuweisung von `ImageMeta`- an `ImageMeta`-Objekte mit `=` sowie die Initialisierung funktioniert, ohne dass wir eine Operatorüberladung und einen Copy-Konstruktor `ImageMeta` geschrieben haben.

  Das liegt daran, dass der vom Compiler automatisch erzeugte `=`-Operator und Copy-Konstruktor für die Basisklasse die bereits implementierten Entsprechungen benutzen; Gleiches gilt für das Standardbibliotheks-Datenelement `comment`.

  Nur wenn wir in `ImageMeta` „neue“ dynamische Datenelemente einführen würden, müssten wir neue Überladungen definieren.

Zuweisungen/Typumwandlungen

- Die Initialisierung/Wertzuweisung an `ImageMeta`- aus `Image`-Objekten mussten wir dagegen ausdrücklich definieren.
- Die andere Richtung, von `ImageMeta` zu `Image`, funktioniert dagegen wieder ohne Implementation (wo tritt dies im Programm auf?)
- Dies gilt allgemein: Ein Basisklassenobjekt kann immer von einem Objekt der abgeleiteten Klasse die Werte übernehmen.

  Die zusätzlichen Datenfelder der abgeleiteten Klasse, hier `comment`, bleiben dabei „auf der Strecke“.

  Andersherum geht das nicht automatisch, weil dann nicht klar wäre, was mit den zusätzlichen Datenfeldern geschehen soll.

UMIT
the health & life sciences university

Beispiel: Quellcode verstehen

→2304-images-class

- Wir betrachten nun das Projekt `2304-images-class`.

  In `main.cpp` wird demonstriert, dass auf eine `ImageMeta`-Variable auch mit einer Referenz vom einfachen Typ `Image` zugegriffen werden kann. Dabei sind natürlich nur Operationen möglich, die in der Klasse `Image` bekannt sind.

- Bei Ansicht des geschriebenen Ausgabebildes fällt aber auf, dass die Kommentare beim Schreiben verloren gegangen sind.

  Grund: Das Bild wurde mittels der `Image`-Referenz und damit der Methode `Image::writepgm` geschrieben, auch wenn das Bild eigentlich vom Typ `ImageMeta` war.

UMIT
the health & life sciences university

Beispiel: Quellcode verstehen

→2305-images-class

- Wir betrachten nun das Projekt 2305-images-class, in dem (fast) dieselbe main.cpp wie zuvor verwendet wird.
- In image.h steht nun

```
virtual bool readpgm (const string filename);
virtual bool writepgm (const string filename) const;}
```

und außerdem

```
virtual ~Image ();
```

In imagemeta.h wurde ergänzt

```
bool readpgm (const string filename) override;
bool writepgm (const string filename) const override;
```

Überschriebene Methoden

- Zuvor hatten wir `readpgm` und `writepgm` für `ImageMeta` einfach nur überschrieben.

  Damit konnte für `ImageMeta`-Objekte eine erweiterte Version dieser Methoden statt derjenigen aus der Basisklasse verwendet werden.

- Wenn eine Referenz – oder ein Zeiger – der Basisklasse für ein Objekt der abgeleiteten Klasse verwendet wird, wird aber in diesem Fall die Methode der Basisklasse benutzt.

  Beim Übersetzen von Quellcode, in dem eine Referenz oder ein Zeiger der Basisklasse verwendet wird, kann der Compiler ja gar nicht wissen, ob zur Laufzeit ein Objekt der Basisklasse oder der abgeleiteten Klasse angesprochen wird!

  Kann man trotzdem mit Zeigern der Basisklasse die korrekte Methode für jedes Objekt aufrufen?

UMIT
the health & life sciences university

Virtuelle Methoden

- Wie wir in `2305-images-class` gesehen haben, lautet die Antwort auf diese Frage *Ja.*

  Man benötigt dazu **virtuelle Methoden.**

- Hierzu wird den Deklarationen der betroffenen Methoden *in der Basisklasse* jeweils das Schlüsselwort `virtual` vorangestellt, also in der Headerdatei der `Image`-Klasse

```
virtual bool readpgm (...);
virtual bool writepgm (...);
```

  – *Das war's schon!*

- *Einmal virtuell – immer virtuell.* Es ist nicht erforderlich, die Methoden in abgeleiteten Klassen erneut als virtuell zu deklarieren.

  Bei der Definition (Implementation) der Methoden darf `virtual` nicht stehen.

UMIT
the health & life sciences university

Virtuelle Methoden

- *Bedeutung:* Für eine virtuelle Methode bestimmt der Compiler beim Übersetzen noch nicht fix, welche Methode (die der Basisklasse oder die einer abgeleiteten Klasse) zur Anwendung kommt.

  Demzufolge bindet auch der Linker die Methodenaufrufe noch nicht an die auszuführenden Methoden.

- Stattdessen wird eine Tabelle der möglichen Kandidaten angelegt, und *zur Laufzeit* wird anhand des tatsächlichen Typs eines Objekts die zu benutzende Methode ausgewählt. Dies heißt **dynamisches Binden.**

- *Konsequenz:* Derselbe Zeiger kann (auch in demselben Programmstatement) zu verschiedenen Zeiten effektiv verschiedene Objekttypen verwalten. Folglich kann derselbe Programmcode mit verschiedenen Objekten je nach Typ Verschiedenes tun.

  So kann die Referenz `newimage` in der `main`-Funktion des Beispiels jedes Bild anders behandeln, also die passende `writepgm`-Methode für das tatsächlich vorliegende Objekt aufrufen.

  Dies nennt man **Polymorphie.**

Virtuelle Destruktoren

Werden dynamisch gebildete Instanzen abgeleiteter Klassen mittels Basisklassen-Zeigern verwaltet, z. B.

```
ptrimage = new ImageMeta;
...
delete ptrimage;
```

so muss der Destruktor virtuell sein!

Dies ist in `1010-images-class` ebenfalls umgesetzt.

- Nur ein virtueller Destruktor stellt sicher, dass die Destruktoren aller Datenelemente einer abgeleiteten Klasse aufgerufen werden.
- Nur so kann `delete` den gesamten Speicher korrekt freigeben.

Destruktoren von Basisklassen sollten daher virtuell sein!