Praxisorientierte Einführung in C++
Lektion: "Vererbung"

Christof Elbrechter

Neuroinformatics Group, CITEC

April 24, 2014

Table of Contents

- Allgemeines
- Aufteilung
- Aufruf von Oberklassenkonstruktoren
- Anmerkung Casts
- Namenskonflikte
- Referenz-Cast
- Importieren von Oberklassen-Bezeichnern
- Slicing
- Nicht-public Vererbung

Allgemeines

- Vererbung hier zunächst im Sinne von Erweiterung einer Klasse
- Modellierung einer *ist-ein*-Beziehung
- z.B.:
  - Katze *ist ein* Säugetier *ist ein* Tier
  - Auto *ist ein* Fahrzeug
  - Dozent *ist ein* Mensch *ist ein* Ding
  - ...
- **Vorteil:** Spezialisierte Objekte können auch überall dort verwendet werden, wo allgemeine verlangt werden

Aufteilung

- In dieser Lektion:
  - Erweitern von Klassen/Strukturen
  - Vererbung vs. Aggregation
  - Wie macht man's richtig?
  - Polymorphie und ihre Umsetzung
- Weitere verwandte Lektionen:
  - C++-cast Funktionen
  - Virtuelle Funktionen mit dem `virtual`-Schlüsselwort
  - `public`, `protected` und `private` Vererbung

Ausgangspunkt

- Eine simple `Point`-Klasse

Point.h

```
class Point{
   int x,y;

public:
  Point(int x=0, int y=0):
     x(x),y(y){}

   int getX() const { return x; }

   int getY() const { return y; }

   void setX(int x){ this->x = x; }

   void setY(int y){ this->y = y; }
};
```

Aggregation

- Nun soll eine Klasse `TextLabel` implementiert werden (hier Aggregation)

TextLabelAggr.h

```
class TextLabelAggr{
   Point pos;
   string l; // Label
 public:
   TextLabelAggr(const Point &pos=Point(),const string &l=""):
     pos(pos),l(l){}

   void setLabel(const string &l){ this->l=l; }
   const string &getLabel() const { return l; }

   // Setter und getter muessen erneut implementiert werden
   void setPos(const Point &pos){ this->pos = pos; }
   const Point &getPos() const{ return pos; }
};
```

Erweitern der Point-Klasse

- 2.Versuch: TextLabel beerbt die `Point`-Klasse
- **Anmerkung:** Hier wird zunächst nur `public`-Vererbung verwendet (... `public Point`)

TextLabel

```
class TextLabel : public Point{
   string label;
 public:
   TextLabel(int x=0, int y=0, const string &label=""):
     Point(x,y),label(label){}

   void setLabel(const string &label){
     this->label = label;
   }
   const std::string &getLabel() const {
     return label;
   }
   // Setter und getter von Point bleiben erhalten
};
```

Aufruf von Oberklassenkonstruktoren

- Oberklassenkonstruktoren müssen zu Beginn der Initialisierungsliste aufgerufen werden

```
TextLabel(int x=0, int y=0, const string &label=""):
   Point(x,y),label(label){}
```

- Falls kein Oberklassenkonstruktor explizit aufgerufen wird, wird automatisch der leere Konstruktor der Oberklasse aufgerufen.

```
TextLabel(int x=0, int y=0, const string &label=""):
   label(label){}
```

- Ruft `Point(int,int)` (mit den Standardargumenten 0,0) auf
- Geht nur, wenn dieser existiert – sonst **muss** existierender Oberklassenkonstruktor aufgerufen werden

Sonstiges ...

- Bei **Mehrfachvererbung**[1] Beerbungsreihenfolge beachten
- **Anmerkung:** Anders als z.B. in Java, können hier keine anderen Konstruktoren aus der eigenen Klasse aufgerufen werden

[1] Ja!, das geht ja in C++

Anmerkung Casts

- Cast = Typumwandlung
- In C++ stehen unterschiedlichste Cast-Operatoren zur Verfügung
- Der *mächtigste* Cast-Operator ist der C-Style-Cast[2] – er bietet **fast** alle Möglichkeiten
- Die Syntax des C-Style-Casts ist analog zu Java-Casts

Beispiele

```
float f = 4.58430437032;
int i = (int)f;
float f2 = ((float)((int)(f*100)))/100; // nur noch 2 Nachkommastellen
```

- Eigentlich gibt es für alle nötigen Fälle einen C++-Cast operator (`static_cast`, `dynamic_cast`, `reinterpret_cast` und `const_cast`)
- Hier zunächst nur der C-Style-Cast

[2] von C übernommen

Namenskonflikte

- Namensbezeichner in abgeleiteten Klassen verstecken gleiche Bezeichner aus der Oberklasse

IntLabel.h

```
class IntLabel : public TextLabel{
  static string str(int i){
    ostringstream s;
    s<<i;
    return s.str();
  }
public:
  IntLabel(int x,int y,int l):TextLabel(x,y,str(l)){}
  void setLabel(int l){
    // setLabel(str(l));  Fehler weil setLabel(string)
    // aus Oberklasse von setLabel(int) verdeckt wird
    // Moeglicher Ausweg: explizite Nennung der Oberklasse
    TextLabel::setLabel(str(l));
  }
};
```

Verdeckte Bezeichner

- Auch *von außen* sind solche Bezeichner plötzlich verdeckt

```
int main(){
   IntLabel l(20,40,1000);
   l.setLabel("hello"); // Fehler versteckt!
}
```

- **1. Ausweg:** expliziter Cast in den Oberklassentypen

```
int main(){
   IntLabel l(20,40,1000);
   ((TextLabel)l).setLabel("hello"); // Gute Idee, geht aber so nicht
}
```

- Problem: Es wird ein Temporary erzeugt! Das ist also äquivalent zu:

```
int main(){
   IntLabel l(20,40,1000);
   TextLabel t = (TextLabel)l;
   t.setLabel("hello"); // Natuerlich keine Auswirkung auf l
}
```

Referenz-Cast

- Wenn schon ein Cast, dann muss es ein Referenz-Cast sein

Beispiel für Referenz-Cast

```
int main(){
   IntLabel l(20,40,1000);
   ((TextLabel&)l).setLabel("hello"); // geht!
}
```

- Das geht, da `l` auch Referenz ist : Also hier nur **reinterpretiert** und nicht **umgewandelt**
- Das ist also äquivalent zu

```
int main(){
   IntLabel l(20,40,1000);
   TextLabel &t = (TextLabel&)l;
   t.setLabel("hello"); // Aendert auch l (da Referenz)
}
```

Importieren von Oberklassen-Bezeichnern

- **2. Ansatz:** Entsprechenden Bezeichner aus der Oberklasse importieren

```
class IntLabel : public TextLabel{
   static string str(int i){...}
 public:
   IntLabel(int x,int y,int l):TextLabel(x,y,str(l)){}
   void setLabel(int l){
     TextLabel::setLabel(str(l));
   }
   TextLabel::setLabel; // sehr einfach!
};

int main(){
   IntLabel l(20,40,1000);
   l.setLabel("hello"); // nun geht es auch so
}
```

Destruktor-Aufrufe

- Destuktoren werden immer automatisch aufgerufen
- Also: Auch Oberklassen-Destruktoren müssen nicht explizit aufgerufen werden – dürfen sie eigentlich auch nicht
- Da immer nur ein Destruktor existiert, bleibt ja auch keine Wahl
- **Oberklassen-Destruktoren werden automatisch (rekursiv) nach dem Destruktor der abgeleiteten Klassen aufgerufen!**

Slicing

- Wie sieht das Binärmuster von C++-Strukturen/Klassen aus?
- Wie verhalten sich Aggregation von Daten und Beerbung von Klassen

Aggregation

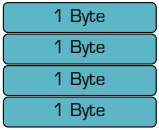
- Simple Strukture `Point`
- Bestehend aus zwei `int`-Members


```
struct Point{
   int x;
   int y;
};
```

- Kleinste Speichereinheit: **1 Byte**
- Speicherbedarf einzelner Datentypen wird in Byte angegeben

Aggregation

```
struct Point{
   int x;
   int y;
};
```



- Ein `int` hat 4 Byte (32Bit-System)

Aggregation

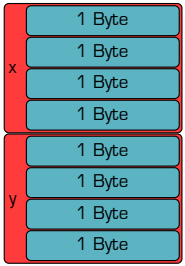
```
struct Point{
   int x;
   int y;
};
```

- **Also:** Ein `int` wird jeweils in einem 4 Byte-Block gespeichert
- Das gilt für `x` ...

Aggregation

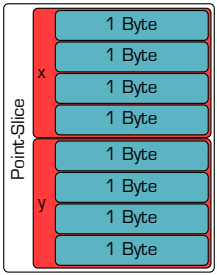
```
struct Point{
   int x;
   int y;
};
```

- **Also:** Ein `int` wird jeweils in einem 4 Byte-Block gespeichert
- ... und für y



Aggregation

```
struct Point{
   int x;
   int y;
};
```

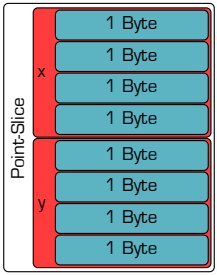


- Zusammen machen die 4 Byte für `x` und die 4 Byte für `y` den gesamten Speicherbedarf von einem `Point`-Objekt aus
- Im Hinblick auf Vererbung nennt man diesen Speicherblock *Point-Slice*

Erweitern von Klassen

- Nun betrachten wird eine simple Erweiterung der `Point`-Klasse

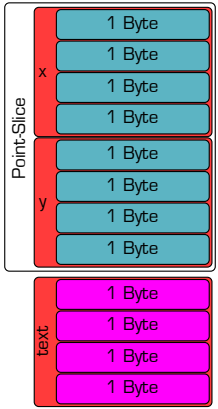
```
struct TextLabel : public Point{
   std::string text;
};
```



- `TextLabel` *ist ein* `Point`
- `TextLabel`-Objekte *beginnen* mit der `Point`-Slice

Erweitern von Klassen

```
struct TextLabel : public Point{
    std::string text;
};
```



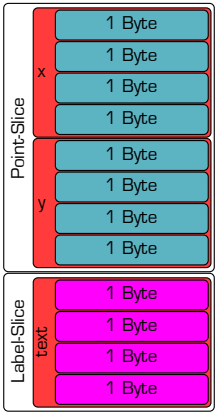
- Instanzen von `std::string` enthalten intern nur einen einzigen Pointer (welcher auf eine komplexere Daten-Struktur zeigt)
- Pointer belegen immer so viel Speicher *wie das Betriebssystem Bit hat* :-)
- In unserem Beispiel:

```
32 Bit = 4 * 8 Bit = 4 Byte
```

Erweitern von Klassen

```
struct TextLabel : public Point{
   std::string text;
};
```

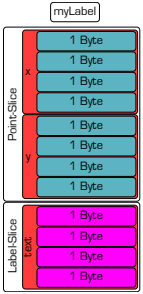
- Damit ist die TextLabel-Slice 4 Byte groß
- Point-Slice und TextLabel-Slice bilden zusammen eine Instanz der TextLabel-Klasse



Slicing bei der Zuweisung und Kopie

- Was passiert bei Zuweisung
  `Point = TextLabel`
  bzw. bei Erstellung
  `Point(const TextLabel&)` ?

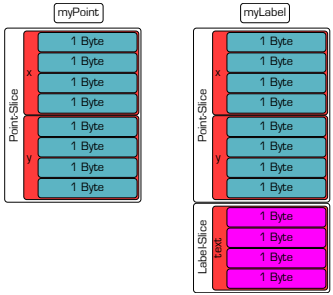
```
TextLabel myLabel(2,2,"Test");
Point myPoint = myLabel;
```



Slicing bei der Zuweisung und Kopie

```
TextLabel myLabel(2,2,"Test");
Point myPoint = myLabel;
```

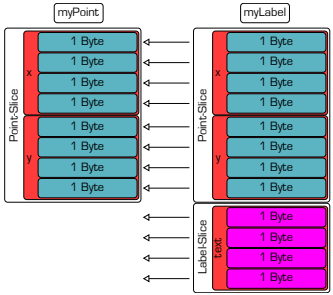
- Bei der Standard-Zuweisung/-Kopie von Objekten, werden die entsprechenden Slices *zueinander ausgerichtet* ...



Slicing bei der Zuweisung und Kopie

```
TextLabel myLabel(2,2,"Test");
Point myPoint = myLabel;
```

- ... und Byte-weise kopiert!
- Was jedoch geschieht mit rvalue-Bytes, die keine Entsprechung auf der lvalue Seite haben?

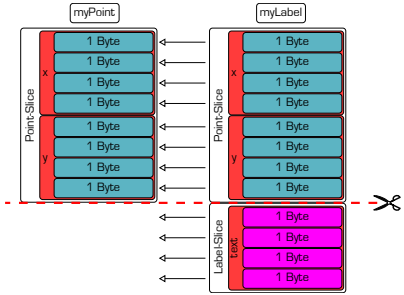


Slicing bei der Zuweisung und Kopie

```
TextLabel myLabel(2,2,"Test");
Point myPoint = myLabel;
```

Achtung

Bei Zuweisung gehen rvalue-Slices verloren, welche im lvalue-Typen nicht vorhanden sind!

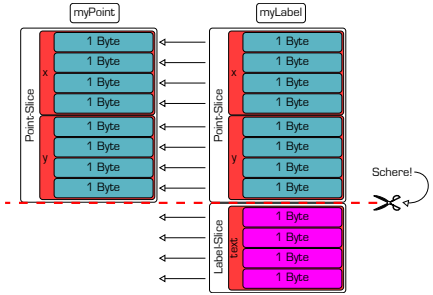


Slicing bei der Zuweisung und Kopie

```
TextLabel myLabel(2,2,"Test");
Point myPoint = myLabel;
```

Achtung

Bei Zuweisung gehen rvalue-Slices verloren, welche im lvalue-Typen nicht vorhanden sind!

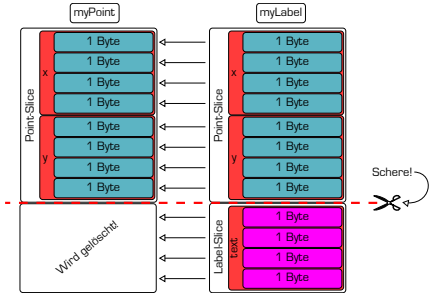


Slicing bei der Zuweisung und Kopie

```
TextLabel myLabel(2,2,"Test");
Point myPoint = myLabel;
```

Achtung

Bei Zuweisung gehen rvalue-Slices verloren, welche im lvalue-Typen nicht vorhanden sind!



Weitere Veranschaulichung

- Gleiches Problem tritt auf bei Arrays auf:

```
Point ps[4] = {
   Point(1,1),
   Point(2,2),
   TextLabel(2,3,"Alles"),
   TextLabel(2,2,"Klar?")
};
```

- ps-Array ist nur groß genug für 4 Objekte des Typs `Point`
- Array besteht ausschließlich aus Points
- Speziellere Objekte werden bei der Zuweisung konvertiert (*gesliced*)

Auswirkungen

- Überschüssige Daten **müssen** *weggeworfen* werden, da lvalue-Instanz nicht genügend Speicher zur Verfügung stellt
- Umgekehrte Zuweisungen (z.B. `TextLabel = Point`) werden **nicht** standardmäßig unterstützt
  - Hier sind ja **zu wenig** Informationen verfügbar!
  - Kann aber implementiert werden
- **Vererbung** umfasst zusätzlich die Möglichkeit, Methoden zu spezialisieren

Auswirkung des *Slicing* auf **Vererbung** in C++

- Vererbung funktioniert in C++ nur vernünftig mit Referenzen und Pointern ($\Rightarrow$ Erklärung: Lektion *Virtuelle Funktionen*)

Nicht-`public` Vererbung

- Bis jetzt haben wir nur `public` Vererbung gesehen
- In C++ gibt es auch `private` und `protected` Vererbung
- Löst die *ist-ein*-Beziehung auf
  - Polymorphismus über Referenzen oder Zeiger funktioniert also nicht mehr
- Ermöglicht Aggregation ohne Elemente

Zugriffsrechte bei Vererbung

Vererbung

Vererbung	public	protected	private	Verhältnis
public	public	protected	private	ist-ein
protected	protected	protected	private	Aggregation
private	private	private	private	Aggregation