

# Programmieren in C++

## SS 2012

Vorlesung 10, Dienstag 10. Juli 2012  
(Vererbung, abstrakte Klassen, virtual)

Prof. Dr. Hannah Bast  
Lehrstuhl für Algorithmen und Datenstrukturen  
Institut für Informatik  
Universität Freiburg

# Blick über die Vorlesung heute

---

## ■ Organisatorisches

- Erfahrungen mit dem 9. Übungsblatt
- Das hier ist die **drittletzte** Vorlesung
- Kurze Wiederholung: Zeiger und \* und &

## ■ Vererbung

- Neben Templates der andere wichtige Mechanismus zur Wiederverwendung von Code / Vermeidung von code duplication
- In dem Zusammenhang wichtig: **virtual** methods
- **Beispiel:** Ein Feld von Objekten verschiedener Klassen mit ein und derselben Oberklasse
- **Übungsblatt 10:** Animation entlang der Umrisse einiger geometrischer Figuren (Kreis, Quadrat, Dreieck)

# Erfahrungen mit dem Ü9 (STL)

---

## ■ Zusammenfassung / Auszüge

Stand 10. Juli 14:00

- Für die meisten zeitlich und überhaupt gut machbar
- Mit der **STL** geht alles viel leichter
- Aber sind die Klassen da drin auch effizient?
- Die "nerds" benutzen **cout**, **ifstream**, **ofstream**, etc ... und wir?
- Muss man den Destruktor immer explizit hinschreiben?
- Probleme mit Zeigern und Unterschied zwischen \* und &
- Wann gibt es die Evaluation? ... **nächste Woche**
- Ist ein Wort auch ein Anagramm von sich selber?
- Umlaute hätten noch erklärt werden sollen
- Eingabe über **cin** geht erst nach **return** weiter

# Erfahrungen mit dem Ü9 (STL)

---

## ■ Fortsetzung ...

- In Vorlesung 1 nicht gesagt, dass 50% vom Projekt nötig
- const correctness in der Vorlage?
- "Ehrlich gesagt weiß ich so langsam nicht mehr was ich hier reinschreiben soll"
- "In der Vorlesung war ich nicht und werde gleich auch wieder nicht gehen"

# Wiederholung: Zeiger und \* und &

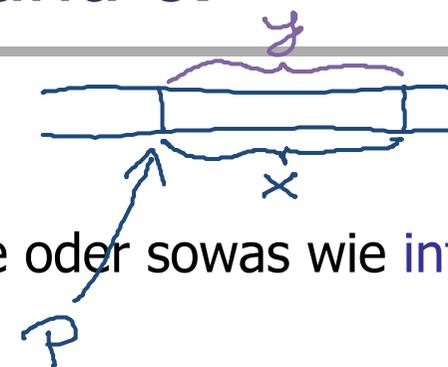
## ■ Also eigentlich ist das ganz einfach

- Sei  $T$  irgendein Typ (egal ob eine Klasse oder sowas wie `int`)
- Dann kann man deklarieren

`T x;` // The name of some memory holding a value of type  $T$ .

`T* p;` // A pointer to the first byte of such a piece of memory.

- Ein `*` **davor** dereferenziert = von  $T^*$  zu  $T$  `x = *p;`
- Ein `&` **davor** macht das Gegenteil = von  $T$  zu  $T^*$  `p = &x;`
- Insbesondere ist also `&*p == p` und `*&x == x`
- Zugriff auf eine Methode oder Membervariable mit Zeiger  
`(*p).some_method()` oder schöner `p->some_method()`
- Ein `&` bei der Deklar. (oder Arg.übergabe) erzeugt einen Alias  
`T& y = x;` // y is now another name for  $x$ .



# Vererbung vs. Templates 1/2

---

- Wofür braucht man Vererbung?
  - Vermeidung von **code duplication** bei zwei oder mehr Klassen die etwas sehr Ähnliches tun
  - Also im Prinzip derselbe Grund wie bei templates
  - Bei welcher Ähnlichkeit benutzt man **templates**?
    - Wenn der Code zweier Klassen bis auf den Typ identisch ist, wie bei `Array<int>` und `Array<char>`
    - Und aus Effizienzgründen ... kurz in der letzten VL angedeutet
  - Bei welcher Ähnlichkeit benutzt man **Vererbung**?
    - Wenn es Methoden / Daten gibt, die gemeinsam sind
    - Aber auch Methoden / Daten, die ganz verschieden sind

# Vererbung vs. Templates 2/2

---

- Warum haben wir erst templates gemacht?
  - Wo doch Vererbung ein Grundprinzip beim objektorientierten Programmierens ist !?
  - Sowohl **templates** als auch **Vererbung** können beliebig kompliziert und knifflig werden
  - In einfachen (praktischen) Anwendungen sind templates sehr simpel, Vererbung aber schon tricky wegen **virtual**
  - Außerdem basiert in der **STL** alles auf templates, und die wollte ich nicht noch später bringen

- Unser Beispiel heute: Ein Feld von "Dingen"

- ... die nichts weiter gemeinsam haben, als dass man sie als `string` darstellen kann

```
class Thing {  
    public:  
        std::string asString() { return "THING"; }  
};
```

- Wir wollen dann nachher sowas schreiben wie

```
std::vector<Thing> things;
```

wobei in `things` ein beliebiger Mix aus ganz verschiedenen Objekten (aus Unterklassen von `Thing`) stehen kann

- Wir definieren jetzt eine **Unterklasse** von **Thing**

- Und zwar ein **Thing**, das eine Zahl enthält

```
class IntegerThing : public Thing {  
    public:  
        IntegerThing(int x) { _value = x; }  
    private:  
        int _value;  
};
```

- Durch das `: public Thing` **erbt** die Klasse `IntegerThing` die Methode `asString` von `Thing`

```
IntegerThing integerThing(5);  
printf("%s\n", integerThing.asString()); // Will print THING.
```

- Wir können die Methode aber überschreiben

- Das nennt man **Polymorphie**

```
class IntegerThing : public Thing {
public:
    IntegerThing(int x) { _value = x; }
    std::string asString() {
        std::ostringstream os; os << _value; return os.str();
    }
private:
    int _value;
};

...

IntegerThing integerThing(5);
printf("%s\n", integerThing.asString()); // Will now print 5.
```

# Vererbung 4/6

---

- Hier noch zwei andere Unterklassen

- ... nach demselben Muster

```
// Thing containing a single character.  
class CharacterThing : public Thing {
```

```
...
```

```
    char _contents;
```

```
};
```

```
// Thing containing a string.
```

```
class StringThing : public Thing {
```

```
...
```

```
    std::string _contents;
```

```
};
```

- Wir würden jetzt gerne sowas machen wie

```
std::vector<Thing> things;  
IntegerThing intThing(5);  
CharacterThing charThing('a');  
StringThing stringThing("doof");  
things.push_back(intThing);  
things.push_back(charThing);  
things.push_back(stringThing);
```

- Das kompiliert zwar ... der Compiler wandelt also offenbar `intThing`, `charThing` und `stringThing` jeweils in ein `Thing` um
- Beim Ausdrucken kommt aber jedes Mal nur `THING`

- Aber das hier funktioniert ohne Weiteres:

```
std::vector<Thing* > things;  
IntegerThing intThing(5);  
CharacterThing charThing('a');  
StringThing stringThing("doof");  
things.push_back(&intThing);  
things.push_back(&charThing);  
things.push_back(&stringThing);
```

- Und jetzt können wir sowas schreiben wie  

```
for (size_t i = 0; i < things.size(); i++)  
    printf("%s\n", things[i]->asString());
```
- Mal schauen, was die Ausgabe davon ist ...

- Ermittlung der korrekten Unterklasse zur Laufzeit
  - Das muss man bei Bedarf in C++ **explizit** angeben:

```
class Thing {  
    public:  
        virtual std::string asString() { return "THING"; }  
};  
  
...  
  
Thing* thing = new IntegerThing(4);  
printf("%s\n", thing->asString()); // Will print 4.
```
  - Durch das **virtual** ermittelt das Programm zur Laufzeit, dass hier die Methode `IntegerString::asString` gemeint ist

- Warum macht das C++ per default nicht immer so?
  - Mit dem `virtual` muss das Programm **zur Laufzeit** nachschauen, auf welches Objekt tatsächlich gezeigt wird, dazu braucht es einen sogenannten `vtable`
  - **Ohne** das `virtual` ist schon beim Kompilieren klar, welche Methode aufgerufen wird, so dass man sich das Nachschauen zur Laufzeit sparen kann
    - Und falls die betreffende Methode `inline` ist, spart man so sogar den Funktionsaufruf

- Achtung: typische Fehlermeldung
  - Wenn man eine virtual Methode aus der Oberklasse (z.B. `Thing`) in der Unterklasse (z.B. `IntegerThing`) deklariert, aber nicht implementiert, bekommt man die Meldung  
... undefined reference to `vtable for IntegerThing'

- Die Klasse Thing macht eigentlich gar nichts
  - Wir haben die Funktion `asString()` dort nur implementiert, weil der Compiler sonst meckern würde
  - Man kann die Methode aber auch **abstrakt** machen, und damit auch die Klasse, das geht einfach so

```
class Thing {  
    public:  
        virtual std::string asString() = 0; // Abstract method.  
};
```

- Jetzt darf man keine Instanzen mehr erzeugen

```
Thing thing; // Will not compile, because of = 0 method.
```

```
Thing* thing; // But a pointer is still fine ... and useful!
```

- Zeiger auf abstrakte Klassen sind aber erlaubt

- Sonst würde unser Beispielprogramm ja gar nicht funktionieren

```
std::vector<Thing*> things;  
IntegerThing intThing(5);  
CharacterThing charThing('a');  
StringThing stringThing("doof");  
things.push_back(&intThing);  
things.push_back(&charThing);  
things.push_back(&stringThing);
```

- Ebenso Referenzen (wird aber nicht empfohlen)

```
const Thing& thing = integerThing; // An alias.
```

# Zeiger auf Unter- / Oberklassen

---

## ■ Automatische Umwandlung

- ... von einem Zeiger auf ein Objekt einer Unterklasse in einen Zeiger auf ein Objekt der Oberklasse:

```
Thing* thing = new IntegerThing(4); // Automatic type cast.
```

```
Thing& thing = integerThing; // Also works for references.
```

- Umgekehrt ist das nicht der Fall

```
Thing* thing = new IntegerThing(4);
```

```
IntegerThing* integerThing = thing; // Will not compile.
```

- Manchmal braucht man das aber, dann muss man **explizit** umwandeln mittels `reinterpret_cast` oder `dynamic_cast`
- Das ist ähnlich wie bei `const_cast` und machen wir in der nächsten Vorlesung

# Virtual destructor

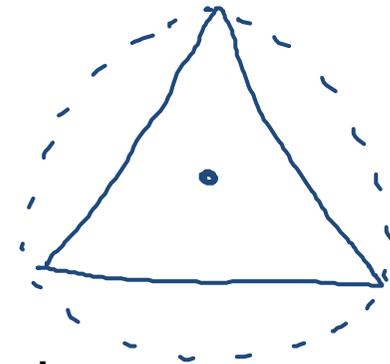
---

- Eine abstrakte Klasse braucht ihn
  - ... sonst meckert der Linter, warum? Hier ist der Grund:  
`Thing* thing = new IntegerThing();`  
...  
`delete thing; // Which destructor gets called now? (*)`
  - Der Zeiger ist vom Typ `Thing*`, deswegen wird bei (\*) ohne Weiteres der Destruktor von `Thing` aufgerufen
  - Wenn jetzt `IntegerThing` intern Speicher alloziert hat, der erst in dessen Destruktor wieder freigegeben wird, wird der Speicher also nicht wieder freigegeben
  - Deklarieren wir für `Thing` einen `virtual destructor`, wird bei (\*) der Destruktor von `IntegerThing` aufgerufen, und alles ist gut

# Hilfestellung zur Animation

## ■ Bewegen eines Zeichens entlang

- ... eines **Kreises**:
  - Siehe [CircleAnimation.cpp](#)
- ... eine **Quadrates**:
  - Ziemlich offensichtlich
- ... eines gleichseitigen **Dreiecks**:
  - Die drei Eckpunkte liegen auch auf einem Kreis
  - Und zwar bei den Winkeln  $0^\circ$ ,  $120^\circ$  und  $240^\circ$
  - Bewegung entlang einer Linie von  $(x_1, y_1)$  nach  $(x_2, y_2)$ 
    - Differenzen  $\Delta x = x_2 - x_1$  und  $\Delta y = y_2 - y_1$  berechnen
    - In jedem Schritt plus  $(\varepsilon \cdot \Delta x, \varepsilon \cdot \Delta y)$



- Vererbung
  - <http://www.cplusplus.com/doc/tutorial/inheritance/>
- Virtuelle Methoden / abstrakte Klassen / Polymorphie
  - <http://www.cplusplus.com/doc/tutorial/polymorphism/>

