Programmieren in C++

SS 2018

Vorlesung 10, Dienstag 3. Juli 2018

(Vererbung)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü9 — STL
  - Diverse Erinnerungen — Treffen Tutor, Evaluation
- Inhalt
  - Grundlagen Vererbung — Oberklassen, Unterklassen
  - Virtuelle Methoden — virtual, pure virtual
  - Casting — static_cast, dynamic_cast
  - **Ü10:** Suche auf einer Wissensdatenbank mit einer Oberklasse und verschiedenen Unterklassen für versch. Arten von Entitäten

    Das letzte Übungsblatt, das nicht schon zum Projekt zählt

Erfahrungen mit dem Ü9 1/2

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Interessante Aufgabe
  - Die STL macht das Leben schon deutlich einfacher

    Aber auch: mehr Nachgucken in der Doku

    Und: STL Fehlermeldungen teilweise sehr verwirrend
  - Die Schnelligkeit von C++ im Vergleich zu Python ist toll
  - Diverse Rückfragen, wie das mit der Evaluation funktioniert

    Siehe Folie 6

Erfahrungen mit dem Ü9 2/2

UNI FREIBURG

- WM-Vorrunde überstehen ohne Tore
  - Fußball ist langweilig / werde Fußball-Fragen boykottieren
  - Wenn alle anderen wegen Doping rausfliegen, dann schon
  - Wenn eine Mannschaft nur noch weniger als sieben Spieler auf dem Feld hat, wird abgebrochen mit 3:0 für die anderen
  - Klar geht das, es können ja alle keine Tore schießen und trotzdem kommen dann zwei weiter
  - Eine Mannschaft kann sogar Weltmeister werden ohne Tore zu schießen (Eigentore)

Treffen mit Ihrem Tutor / Ihrer Tutorin

UNI FREIBURG

- Erinnerung
  - Die meisten haben sich inzwischen schon getroffen bzw. den Termin für das Treffen schon ausgemacht
  - Ein paar wenige haben auf die (wiederholten) Mails von noch gar nicht reagiert
  - Bitte dringend melden, denn das Treffen ist notwendige Bedingung für das erfolgreiche Bestehen der Veranstaltung
  - Falls Sie noch gar keine Mail dazu bekommen haben, melden Sie sich bitte bei Ihrem Tutor / Ihrer Tutorin

Offizielle Evaluation der Veranstaltung

UNI FREIBURG

- Läuft über das zentrale **EvaSys** der Uni
  - Sie sollten letzte Woche schon (Mittwoch, 27. Juni) eine Mail vom System bekommen haben

    Falls nicht, bitte kurz auf dem Forum Bescheid geben, wir haben dafür ein Unterforum "Evaluation" eingerichtet
  - Nehmen Sie sich bitte Zeit und füllen Sie den Bogen sorgfältig und gewissenhaft aus

    Sie haben soviel Zeit in die Vorlesung investiert, dann können Sie auch 20 Minuten für die Evaluation aufwenden

    Sie bekommen außerdem 20 Punkte dafür, die die Punkte vom schlechtesten ÜB ersetzen, oder +10 Punkte für das Projekt, was immer für Sie günstiger ist … siehe V1
  - Uns interessieren besonders die Freitextkommentare

Vererbung Grundlagen 1/9

UNI FREIBURG

- Motivation, Unterschied zu Templates
  - Wiederverwendung von Code, wenn zwei oder mehr Klassen etwas sehr Ähnliches tun

    Also ähnlicher 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, wie in der letzten VL erklärt
  - Bei welcher Ähnlichkeit benutzt man **Vererbung**?

    Wenn es Methoden / Daten gibt, die gemeinsam sind

    Aber auch Methoden / Daten, die ganz verschieden sind

Vererbung Grundlagen 2/9

UNI FREIBURG

- Warum Vererbung erst so spät
  - Wo doch Vererbung ein Grundprinzip beim objekt-orientierten Programmieren ist
  - Grund: 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

Vererbung Grundlagen 3/9

UNI FREIBURG

- Unser Beispiel heute: Ein Feld von "Dingen"
  - Einzige Gemeinsamkeit: eine toString Methode

```
class Thing {
 public:
  std::string toString() { 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

Vererbung Grundlagen 4/9

UNI FREIBURG

- Unterklassen
  - Eine **Unterklasse** von Thing, die eine Zahl enthält

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

  - Durch das : public Thing **erbt** die Klasse IntegerThing alle Methoden und Membervariablen von Thing

    In dem Fall ist das nur die Methode toString

```
IntegerThing it(5);
printf("%s\n", it.toString());  // Prints THING.
```

Vererbung Grundlagen 5/9

UNI FREIBURG

- Polymorphie
  - Man kann die Methoden aus der Oberklasse aber in der Unterklasse überschreiben, z.B.

```
class IntegerThing : public Thing {
   public: IntegerThing(int x) { _value = x; }
   public: std::string toString() { return std::to_string(_value); }
   private: int _value;
};

...

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

Vererbung Grundlagen 6/9

UNI FREIBURG

- Weitere Unterklassen
  - Nach dem selben Muster können wir jetzt weitere Unterklassen definieren

```
// Thing containing a string.
class StringThing : public Thing {
  ...
  std::string _contents;
};
```

  - Die Unterklassen untereinander können dabei ganz verschieden sein … einzige Gemeinsamkeit:

    Was sie von der Oberklasse erben

    Mit **: public** Zugriff auf alles aus der Oberklasse, mit private oder : protected nicht unbedingt, siehe Folie 26

Vererbung Grundlagen 7/9

UNI FREIBURG

- Verwendung, Versuch 1
  - Ein Feld von verschiedenen Dingen

```
std::vector<Thing> things;
IntegerThing thing1(42);
StringThing thing2("doof");
things.push_back(thing1);
things.push_back(thing2);
for (auto& thing : things) { cout << thing.toString(); }
```

  - Das kompiliert auch ohne Probleme
  - Der Compiler wandelt also offenbar Objekte vom Typ IntegerThing und StringThing automatisch in Thing um
  - Allerdings kommt beim Ausdrucken immer nur THING

Vererbung Grundlagen 8/9

UNI FREIBURG

- Verwendung, Versuch 2
  - Dasselbe nur mit Zeigern

```
std::vector<Thing*> things;
IntegerThing* thing1 = new IntegerThing(42);
StringThing* thing2 = new StringThing("doof");
things.push_back(thing1);
things.push_back(thing2);
for (auto thing : things) { cout << thing->toString(); }
```

  - Das kompiliert ebenfalls ohne Probleme
  - Auch die Zeiger IntegerThing* und StringThing* werden also automatisch in Thing* umgewandelt
  - Es kommt aber immer noch nur THING

Vererbung Grundlagen 9/9

UNI FREIBURG

- Verwendung, Versuch 3
  - Dasselbe wie auf der Folie vorher (mit Zeigern), aber in der Deklaration von Thing schreiben wir jetzt

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

  - Jetzt erst kommt die erwartete Ausgabe

    Das klappt aber nicht mit dem Code von Versuch 1 (ohne Zeiger), selbst mit dem virtual oben

    Warum wird auf den nächsten Folien erklärt

Virtuelle Methoden 1/8

UNI FREIBURG

- Grundprinzip
  - Die Frage ist, welche toString Methode in diesem Code aufgerufen werden soll

    ```
    std::vector<Thing*> things;
    ...
    for (auto thing : things) { std::cout << thing->toString(); }
    ```

  - Dabei wichtig zum Verständnis:

    Beim Kompilieren kann man den Typ der Objekte, auf den die Zeiger in dem Feld zeigen, im Allg. **nicht** wissen

    Es könnte zum Beispiel von der Eingabe abhängen, welches Objekt von welchem Typ ist ... siehe Ü10

Virtuelle Methoden 2/8

UNI FREIBURG

- Ohne virtual
  - Die Frage ist, welche toString Methode in diesem Code aufgerufen werden soll

    ```
    std::vector<Thing*> things;
    ...
    for (auto thing : things) { cout << thing->toString(); }
    ```

  - **Ohne** das virtual in der Deklaration von Thing::toString wird die Entscheidung vom **Compiler** getroffen

    Und der kann nur entscheiden, die Thing::toString Methode aufzurufen, weil er nicht mehr Infos hat

    Deswegen wird dann nur THING gedruckt

Virtuelle Methoden 3/8

UNI FREIBURG

- Mit virtual
  - Die Frage ist, welche toString Methode in diesem Code aufgerufen werden soll

    ```
    std::vector<Thing*> things;
    ...
    for (auto thing : things) { cout << thing->toString(); }
    ```

  - Mit dem virtual in der Deklaration von Thing::toString wird die Entscheidung zur Laufzeit getroffen
  - Dazu wird für die Klasse ein sogenannter **vtable** angelegt

    Der enthält die Adressen aller Funktionsaufrufe dieser Klasse, die nicht schon beim Kompilieren feststehen

    Für die anderen ohne wird der vtable nicht benutzt

Virtuelle Methoden 4/8

UNI FREIBURG

- Warum die Unterscheidung
  - Die Benutzung des **vtable** kostet Platz und Laufzeit, die meisten Compiler realisieren das so:

    Der Compiler fügt der Klasse (heimlich) eine Member-variable hinzu, die ein Zeiger auf den vtable der Klasse ist

    Zu jedem Konstruktor (der Oberklasse und Unterklassen) wird Code hinzugefügt, der diesen Zeiger geeignet setzt

    Siehe https://en.wikipedia.org/wiki/Virtual_method_table
  - Es ist ja ein Grundprinzip von C++ (anders als z.B. in Java) nichts in die Sprache einzubauen was Performanz kostet, ohne dass man etwas dagegen machen kann

    Der Preis dafür ist eine komplexere Sprache

Virtuelle Methoden 5/8

UNI FREIBURG

- Typische Fehlermeldung
  - Typischer Fall: eine Methode ist in der Oberklasse virtual deklariert … und auch in der Unterklasse deklariert

    Ob in der Unterklasse auch virtual spielt hierbei keine Rolle, das wäre erst wieder für eine Unterklasse der Unterklasse von Bedeutung
  - Die Methode der Unterklasse wird **nicht** implementiert

    Weil vergessen oder sich beim Methodennamen vertippt
  - Dann kommt eine Fehlermeldung wie

    ... undefined reference to `vtable for IntegerThing'

Virtuelle Methoden 6/8

UNI FREIBURG

- Virtueller Destruktor
  - Aus demselben Grund wie auf Folie 20, braucht man manchmal auch einen virtuellen Destruktor, Beispiel:

    ```
    Thing* thing = new IntegerThing();
    ...
    delete thing;
    ```

  - Wenn Thing **keinen** virtuellen Destruktor hat, wird beim delete der Default-Destruktor von Thing aufgerufen

    Der Destruktor von IntegerThing wird dann **nicht** aufgerufen und dadurch wird evtl. Speicher nicht freigegeben

    Hat Thing dagegen einen virtuellen Destruktor wird der Destruktor von IntegerThing aufgerufen und alles ist gut

Virtuelle Methoden 7/8

UNI FREIBURG

- Abstrakte Klassen, Motivation
  - In Thing haben wir die Methode toString() nur deshalb 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.
```

Virtuelle Methoden 8/8

UNI FREIBURG

- Abstrakte Klassen, Verwendung
  - Ein Zeiger auf eine abstrakte Klasse ist aber erlaubt

    Sonst könnte man gar nichts mit so einer Klasse machen

    Unser Beispielprogramm von vorhin (Folie 13) funktioniert also auch, wenn Thing abstrakt ist
  - Eine abstrakte Referenz ist auch erlaubt

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

    Aber nicht empfehlenswert, bei sowas immer Zeiger benutzen, dann klarer was hinter den Kulissen passiert

Zugriffsrechte 1/3

UNI FREIBURG

- Vererbung von privaten Membervar. und -methoden
  - Unterklassen haben keinen direkten Zugriff auf **private** Membervariablen oder –methoden der Oberklasse
  - Sie werden aber trotzdem vererbt und Methoden der Oberklasse dürfen (natürlich) auf sie zugreifen

```
class Entity {
   private: char* _name;
   public: getName() { return _name; }
};

class Person : public Entity {
   public print() { printf(_name);  }       // Does not compile.
   public print() { printf(getName(); }   // This is fine.
}
```

Zugriffsrechte 2/3

UNI FREIBURG

- Protected
  - Es gibt auch noch **protected**, das wirkt außerhalb der Klasse wie private, wird aber (anders als private) vererbt

```
class Entity {
   protected: char* _name;
   public: getName() { return _name; }
};

class Person : public Entity {
   public: print() { printf(_name); }       // This is fine now.
};

Entity entity; printf(entity._name);       // Does not compile.
Person person; printf(person._name);  // Neither does this.
```

Zugriffsrechte 3/3

UNI FREIBURG

- Maximale Zugriffsrechte bei der Vererbung
  - Die Angabe hinter dem Doppelpunkt der Klassendeklaration setzt ein Limit für die maximalen Zugriffsrechte

```
class StringThing : protected Thing {
    // Public members from Thing are protected here.
    // Protected members from Thing are protected here.
    // Private members from Thing are not visible here.
}

class StringThing : private Thing {
    // Public members from Thing are private here.
    // Protected members from Thing are private here.
    // Private members from Thing are not visible here.
}
```

Zeigerkonvertierung 1/3

UNI FREIBURG

- Zeiger Unterklasse → Zeiger Oberklasse
  - Ein Zeiger auf eine Unterklasse wird vom Compiler anstandslos in einen Zeiger auf die Oberklasse konvertiert

    ```
    Thing* thing = new IntegerThing(42);     // Works.
    ```

    Das braucht man in der Praxis auch ziemlich oft
  - Geht auch mit Referenzen

    ```
    Thing& thing = integerThing;             // Works.
    ```

    Das benutzt man so kaum, sondern fast immer mit Zeigern

Zeigerkonvertierung 2/3

UNI FREIBURG

- Zeiger Oberklasse → Zeiger Unterklasse
  - Umgekehrt ist das nicht der Fall

```
Thing* t;
IntegerThing* it = t;  // Will not compile.
```

  - Will man es trotzdem, muss man **explizit** umwandeln

```
Thing* t = new IntegerThing(42);
IntegerThing* it = dynamic_cast<IntegerThing*>(t);
```

    Dabei schaut dynamic_cast zur Laufzeit, ob der Typ auch wirklich stimmt, wenn nicht, wird NULL zurück gegeben

    Mit static_cast wird ohne Typcheck konvertiert

    Beispiel wo man das braucht: C++ SS 2010, Vorlesung 11

Zeigerkonvertierung 3/3

STOP
Nein
Böse
Nicht tun!

UNI FREIBURG

- Beliebige Zeigerkonvertierung
  - Mit **reinterpret_cast** kann man Zeiger auf **beliebige** Typen ineinander umwandeln

```
class Xyz { … };                          // Some class.
int m = sizeof(Xyz);                      // Size of an Xyz object.
char* p = new char[m];                    // Space for an Xyz object.
Xyz* q = p;                               // Does not compile.
Xyz* q = reinterpret_cast<Xyz*>(p);       // This does.
```

  - Das braucht man in der Anwendung nur bei sehr maschinennahen Anwendungen, etwa bei Treibersoftware

    Wenn man das in gewöhnlichem Code macht, braucht man es entweder nicht, oder der Code ist schlecht designed

Aufruf Methoden Oberklasse 1/3

UNI FREIBURG

- Konstrukturen & Destruktoren
  - Implizit ruft jeder Konstruktor einer Unterklasse zu Beginn den Default-Konstruktor der Oberklasse auf

    Der ruft dann, bei mehrstufiger Vererbung, wiederum den Default-Konstruktor seiner Oberklasse auf, usw.
  - Bei den Destruktoren dito, in umgekehrter Reihenfolge

    Also erst der Destruktor der Unterklasse, dann der von der Oberklasse, dann der von deren Oberklasse, usw.

Aufruf Methoden Oberklasse 2/3

UNI FREIBURG

- Konstrukturen & Destruktoren
  - Wenn man will, dass ein anderer Konstruktor der Oberklasse aufgerufen wird, geht das so

```
class MyThing : public StringThing { … }

MyThing::MyThing(std::string s) : StringThing(s) {
   …
}
```

  - Will man mehrere solcher Konstruktoren aufrufen (z.B. für mehrere Argumente), dann durch Komma trennen

    Konstruktor der Oberklasse explizit im Code aufrufen geht **nicht**, wenn man das braucht, eigene Methode schreiben

Aufruf Methoden Oberklasse 3/3

UNI FREIBURG

- Methoden der Oberklasse
  - Andere Methoden der Oberklasse kann man ganz normal mit Ihrem voll-qualifizierten Namen aufrufen

```
class Person : public Entity { … };
…
// Show information about a person.
void Person::show() {
   Entity::show();        // First call method from Entity.
   …;                     // Arbitray additional code.
}
```

  - So etwas wie "super" in Java gibt es in C++ nicht, weil eine Klasse mehrere Oberklassen haben kann

    Bei multipler Vererbung, die benutzt man aber in der Praxis selten

Literatur / Links

UNI FREIBURG

- Vererbung
  - http://www.cplusplus.com/doc/tutorial/inheritance/
- Polymorphie
  - http://www.cplusplus.com/doc/tutorial/polymorphism/