Programmieren in C++

SS 2022

Vorlesung 10, Dienstag 5. Juli 2022

(Vererbung)

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

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü9 — STL
  - Offizielle Evaluation — Aufgabe 0 auf dem Ü10
  - Vorgucker auf das Projekt — V11 erst wieder am 19. Juli

- Inhalt
  - Grundlagen Vererbung — Oberklassen, Unterklassen
  - Virtuelle Methoden — virtual, override, pure virtual
  - Mehr zur Vererbung — Aufruf Methoden der Oberklasse, Zugriffsrechte, Zeigerkonvertierung

Ü10: Eine Unterklasse MockTerminalManager, die einen Bildschirm simuliert und mit der man drawHeatMap testen kann

Erfahrungen mit dem Ü9 1/4

UNI FREIBURG

- Feedback zur Aufgabe
  - Für die meisten wieder schwieriger und etwas mehr Arbeit

    "Schwieriger als die letzten ÜB, dafür aber auch spannender"

    "Viel Spaß, aber am Anfang etwas überwältigt von der STL"
  - Manchen hat die Freiheit bei der Aufgabenstellung gefallen, andere hätten sich mehr Vorgaben gewünscht

    "Habe es genossen, wieder freier programmieren zu können"
  - Visualisierung war ein schönes Erfolgserlebnis → Demo Musterlösung

    Einige hatten Probleme mit dem TerminalManager unter WSL (Windows), die konnten aber über das Forum gelöst werden
  - Deutlicher "negativity bias" bei denen, die (zu) spät mit dem Übungsblatt angefangen haben

Erfahrungen mit dem Ü9 2/4

UNI FREIBURG

- Noch ein paar Zitate

  "Stolz, dass ich endlich dieses 'inkrementelle Programmieren' sehr radikal durchgezogen habe, wäre sonst untergegangen"

  "Freue mich, dass mein Code endlich so aussieht wie von den coolen Leuten auf Stack Overflow, bei denen ich mich immer über ihr std:: und << gewundert habe."

  "Nach dem Übungsblatt verstehe ich definitiv, warum die STL erst jetzt zugelassen wurde; die Komplexität der Fehlermeldungen ist doch nochmal etwas ganz anderes."

  "Aus privaten Gründen hatte ich keine Zeit für dieses Übungsblatt; Zeitaufwand: 2 Minuten ;)"

Erfahrungen mit dem Ü9 3/4

UNI FREIBURG

- Übergeordnetes Thema: Anspruchshaltung
  - Man kann es so machen

    "Das mit dem TerminalManager (Fehler von euch) hat echt viel Zeit in Anspruch genommen, bitte in Zukunft vermeiden"

    "Große Probleme mit der Hashfunktion, kam meiner Meinung nach auch sehr kurz in der VL"
  - Oder so:

    "Ich konnte den TerminalManager nicht richtig ausführen. Zum Glück ist das Vorlesungsteam aber extrem engagiert und opfert sogar seine Freizeit, um den Student*innen zu helfen."

    "Alles womit ich Probleme hatte, wurde im Forum thematisiert (hauptsächlich das mit der eigenen Hashfunktion)"

Erfahrungen mit dem Ü9 4/4

UNI FREIBURG

- OpenStreetMap (OSM)
  - Initiiert von Steve Coast in 2004, inspiriert von Wikipedia

    Wie Wikipedia ein kollaboratives Projekt, bei dem User einfach Daten hochladen können
  - Die Datenfüllen und "Coverage" ist absolut beeindruckend

    7.8 Milliarden Punkte, 0.9 Milliarden Wege und Flächen Stand 03.07.2022

    "OSM hat es echt alles, ich bin ein großer Fan davon"
  - Bei uns an der Professur arbeiten wir viel mit OSM Daten

    https://www.openstreetmap.org

    https://qlever.cs.uni-freiburg.de/osm-germany/sdXK74

Offizielle Evaluation der Veranstaltung

UNI FREIBURG

- Läuft über das zentrale **EvaSys** der Uni
  - Sie sollten dazu am Montag (4. Juli) 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 seien Sie ehrlich und fair

    Sie haben soviel Zeit in die Veranstaltung 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
  - Für uns sind insbesondere die Freitextkommentare interessant

Projekt 1/2

UNI FREIBURG

- Vorgucker
  - Gegenstand des Projekts wird ein Spiel sein, und zwar:

    https://nerdlegame.com
  - Es wird (wie immer) drei Projekte zur Auswahl geben

    Projekt 1: Das Spiel implementieren (mit ncurses)

    Gut machbar, Arbeit hält sich in Grenzen

    Projekt 2: Einen Solver für das Spiel

    Erfordert: Spaß am Knobeln und effizientem Code … Grafik und Logik ist per Bibliothek vorgegeben (zum Debuggen)

    Projekt 3: Ein frei wählbares Projekt Ihrer Wahl

    Nur für die, die sich schon sehr sicher im Umgang mit C++ fühlen und keine Probleme mit den Übungsblättern hatten

Projekt 2/2

UNI FREIBURG

- Ein paar Details
  - Projekt 1 wird im Wesentlichen drei Teile haben

    1. Eine zufällige Gleichung ermitteln
    2. Die Spiellogik implementieren
    3. Das Malen des Spielstands implementieren

  - Projekt 2 wird im Wesentlichen zwei Teile haben

    1. Über alle noch möglichen Lösungen iterieren
    2. Das Iterieren geschickt und effizient gestalten

  - Bedingungen für Projekt 3 (Thema ihrer Wahl, für Fortgeschrittene)

    Das verhandeln Sie mit Ihrem Tutor oder Ihrer Tutorin
    (und im Zweifelsfall mit Johannes Kalmbach und mir)

  - Alles Weitere in der V11 am **19. Juli**

Nächste Woche keine Vorlesung

UNI FREIBURG

- Wie angekündigt!
  - Einerseits **als Ausgleich** für die Vorlesungen, die etwas länger gedauert haben als geplant
  - Andererseits **als Verschnaufpause** und damit Sie neben dem Ü10 auch schon über das Projekt nachdenken können

    Wir empfehlen, das Projekt noch während der Vorlesungszeit fast oder ganz fertig zu stellen

    Ab jetzt noch vier Wochen, ab der V11 noch zwei Wochen
  - Die offizielle Deadline ist Dienstag, 9. August 12:00 Uhr

    Auf begründeten Antrag verlängerbar (zum Beispiel falls das bei Ihnen mit einer intensiven Prüfungsphase kollidiert)

Grundlagen Vererbung 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**?

    Zum Beispiel wenn der Code zweier Klassen bis auf den Typ identisch ist, wie bei **Array<int>** und **Array<char>**

    Typentscheidung beim Kompilieren, Implemt. in .h Dateien
  - Bei welcher Ähnlichkeit benutzt man **Vererbung**?

    Zum Beispiel bei verschiedenen Implementierungen eines gemeinsamen Interfaces, wie beim Ü10

    Typentscheidung zur Laufzeit, Implemt. dazulinken möglich

Grundlagen Vererbung 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

Grundlagen Vererbung 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

Grundlagen Vererbung 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);
std::cout << it.toString());  // Prints THING.
```

Grundlagen Vererbung 5/9

UNI FREIBURG

- Polymorphie
  - Man kann die Methoden aus der Oberklasse aber in der Unterklasse überschreiben, zum Beispiel

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

...

IntegerThing it(5);
std::cout << it.toString());  // Will now print 5.
```

Grundlagen Vererbung 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

Grundlagen Vererbung 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 (Thing& 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

    Es wird aber nur "THING" ausgegeben

Grundlagen Vererbung 8/9

UNI FREIBURG

- Verwendung, Versuch 2
  - Dasselbe nur mit Zeigern

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

  - Das kompiliert ebenfalls ohne Probleme
  - Auch die Zeiger IntegerThing* und StringThing* werden also automatisch in Thing* umgewandelt

    Es wird wieder nur "THING" ausgegeben

Grundlagen Vererbung 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 1/10

UNI FREIBURG

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

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

  - Dabei wichtig zum Verständnis:

    Der **Compiler** kann den Typ der Objekte, auf den die Zeiger in dem Feld zeigen, im Allgemeinen **nicht** im Voraus kennen

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

Virtuelle Methoden 2/10

UNI FREIBURG

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

    ```
    std::vector<Thing*> things;
    ...
    for (Thing* 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 3/10

UNI FREIBURG

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

    ```
    std::vector<Thing*> things;
    ...
    for (Thing* 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

    Enthält die Adressen aller Methoden der Klasse, die nicht schon beim Kompilieren feststehen … im Beispiel: eine

    Für die anderen Methoden wird der vtable **nicht** benutzt

Virtuelle Methoden 4/10

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, das Performanz kostet, ohne dass man etwas dagegen machen kann

    Der Preis dafür ist eine komplexere Sprache

Virtuelle Methoden 5/10

UNI FREIBURG

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

    Ob sie auch in der Unterklasse **virtual** deklariert ist, spielt hierbei keine Rolle, das wäre erst wieder für eine Unterklasse der Unterklasse von Bedeutung
  - … und die Methode wird in der Oberklasse oder der Unterklasse **nicht** implementiert

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

    … undefined reference to `vtable for IntegerThing'

Virtuelle Methoden 6/10

UNI FREIBURG

- Override
  - Wenn man eine virtuelle Methode einer Oberklasse überschreibt, sollte man das keyword **override** verwenden

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

class StringThing {
  public: std::string asString() override { return contents_; }
  private: std:string contents_;
};
```

  - Mit der Option –Wsuggest-override warnt einen der Compiler, wenn man das override an so einer Stelle weglässt
  - Ein override **ohne** virtual in der Oberklasse ist ein Fehler

Virtuelle Methoden 7/10

UNI FREIBURG

- Virtueller Destruktor
  - Der Destruktor einer Oberklasse sollte in der Regel virtual sein, zum 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 8/10

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 toString() = 0;  // Abstract method.
};
```

  - Jetzt darf man keine Instanzen mehr erzeugen

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

Virtuelle Methoden 9/10

UNI FREIBURG

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

    Sonst könnte man nichts mit so einer Klasse machen

    Unser Beispielprogramm von "Versuch 2" (Folie 18) funktioniert also auch, wenn Thing abstrakt ist
  - Eine abstrakte Referenz ist auch erlaubt

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

    Aber nicht empfehlenswert! Bei so etwas immer Zeiger benutzen, damit klarer ist, was hinter den Kulissen passiert

Virtuelle Methoden 10/10

UNI FREIBURG

- Warum funktioniert virtual nur mit Zeigern?
  - Der Code von "Versuch 1" gibt auch dann zweimal "THING" aus, wenn toString in Thing als virtual deklariert ist

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

Man **muss** sich hier zur Compile-Zeit für einen Typ entscheiden, weil IntegerThing und StringThing verschieden groß sind, und da bleibt nur die Oberklasse Thing als einzig sinnvolle Wahl

Ein Zeiger ist immer gleich groß, da kann die Entscheidung, auf welchen Typ er zeigt, auch zur Laufzeit getroffen werden

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

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

class SpecialStringThing : public StringThing { … }

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

  - Ohne das StringThing(s) nach dem Doppelpunkt würde hier der Defaultkonstruktor von StringThing aufgerufen

    Es gäbe dann **im** Code der Funktion auch keine Möglichkeit mehr, einen anderen Konstruktor aufzurufen

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 Thing { public: virtual std::string toString() { … } };

class StringThing : public Thing {
   public std::string toString() override {
     return Thing::toString() + " " + contents_;
   }
   std::string contents_;
}
```

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

    Multiple Vererbung in der Praxis allerdings selten

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: const char* name_;
   public: void getName() { return name_; }
};

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

Zugriffsrechte 2/3

UNI FREIBURG

- Protected
  - Es gibt auch noch **protected**, das wirkt außerhalb der Klasse wie private, aber die Unterklasse hat Zugriff darauf

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

class Person : public Entity {
   public: void 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 accessible 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 accessible 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, siehe den Code von Folie 18
  - Das 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 **nullptr** zurück gegeben

    Wenn man diesen Laufzeitcheck nicht möchte: –fno-rtti

    Mit static_cast wird ohne Typcheck konvertiert

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

Literatur / Links

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