Programmieren in C++

SS 2011

Vorlesung 10, Mittwoch 13. Juli 2011

(Vererbung, abstrakte Klassen, virtual)

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 9. Übungsblatt
  - Dies ist die **viertletzte** Vorlesung
  - Treffen mit Ihrem Tutor, siehe **Ü10** !!!
- Vererbung
  - Neben Templates der andere wichtige Mechanismus zur Wiederverwendung von Code / Vermeidung von code duplication
  - In dem Zusammenhang wichtig: virtual methods
  - **Ü10:** Implementieren Sie 2 – 3 Sortieralgorithmen (einer davon basierend auf std::sort) in Unterklassen einer gemeinsamen Basisklasse, und lassen Sie sie gegeneinander antreten

Erfahrungen mit dem Ü9 (STL)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 13.7 15:47
  - STL hilfreich und Benutzung macht Spaß
  - Für das Parsen hat die STL aber nicht so viel gebracht
  - Für den calculator auch nicht
  - Schön zu sehen wieviel sich seit dem Ü4 getan hat
  - asString erleichert das Testen sehr
  - Danke für das Erholungsblatt!
  - Ü5 statt Ü4 als Basis wäre passender gewesen
  - Warum machen wir keine Iteratoren?
  - Gut: sowohl (erst) selber machen als auch (dann) Bibliotheken
  - Warum die STL nicht früher?
  - Test sind toll! (ernst und nicht ironisch gemeint)

Treffen mit Ihrem Tutor

UNI FREIBURG

- Jetzt wird es ernst
  - Wer sich bisher noch keinmal getroffen hat, **unbedingt** bis zur **nächsten Woche** einen Termin ausmachen
  - Wer trotz wiederholter E-Mail nicht reagiert bekommt **keine** ECTS Punkte für den Kurs
  - Falls Sie sich treffen wollen, aber es Ihrer Meinung nach nicht an Ihnen liegt, dass es nicht klappt, E-Mail an **mich**

Vererbung vs. Templates 1/2

UNI FREIBURG

- Wofür braucht man Vererbung?
  - Möglichst gute Wiederverwendung von Code bzw. 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 ... dazu (vielleicht) mehr in einer der letzten Vorlesungen
  - 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

UNI FREIBURG

- 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

Vererbung 1/6

UNI FREIBURG

- 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

Vererbung 2/6

UNI FREIBURG

- Wir definieren jetzt eine **Unterklasse** von Thing
  - Und zwar ein Thing, dass 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.
```

Vererbung 3/6

UNI FREIBURG

- 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;
};
```

Vererbung 5/6

UNI FREIBURG

- 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 aber nicht, warum?
  - Weil wir dazu intThing, charThing und stringThing jeweils in ein Thing umwandeln müssten
  - Das geht aber nicht, es sei denn wir schreiben eine extra Funktion dafür (und zwar einen copy constructor !)

Vererbung 6/6

- 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 ...

Virtual Methods 1/3

UNI FREIBURG

- Wir hätten gerne
  - … das der Compiler für jedes Thing weiß, von welchem Typ es ist; das muss man C++ **explizit mitteilen**:

```
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

Virtual Methods 2/3

UNI FREIBURG

- Na also geht doch, warum dann nicht immer so?
  - **Mit** dem virtual muss das Programm **zur Laufzeit** nachschauen, auf welches Objekt tatsächlich gezeigt wird, dazu braucht es einen sog. vtable
  - **Ohne** das virtual ist schon beim Kompilieren klar, welche Methode aufgerufen wird, so dass man sich die Indirektion zur Laufzeit sparen kann

    (Anmerkung für die nerds: und unter Umständen sogar den Funktionsaufruf, wenn man die Methode inlinen kann)

Virtual Methods 3/3

UNI FREIBURG

- 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'

Abstrakte Methoden / Klassen 1/2

UNI FREIBURG

- 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!
```

Abstrakte Methoden / Klassen 2/2

UNI FREIBURG

- 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;  // Alias.
```

Zeiger auf Unter- / Oberklassen

UNI FREIBURG

- 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

UNI FREIBURG

- Eine abstrakte Klasse braucht ihn
  - … sonst meckert der Linter, warum? Hier ist der Grund:

```
Thing* thing = new IntegerThing();                    (1)
...
delete thing;  // Which destructor gets called now?   (2)
```

  - Der Zeiger ist vom Typ Thing*, deswegen wird bei (2) 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 (2) der Destruktor von IntegerThing aufgerufen, und alles ist gut

Zeitmessung

UNI FREIBURG

- Brauchen Sie für das Ü10
  - Das geht entweder mit clock und CLOCKS_PER_SECOND

    man 3 clock

    Da gibt es aber Probleme auf manchen Rechnern, wegen schlechter Auslösung / ungenauer Messung
  - Oder mit der Funktion gettimeofday

    man 3 gettimeofday

    Dass misst dann aber die tatsächlich vergangene Zeit (nicht nur die CPU-Zeit) was aber oft das ist was man will

Literatur / Links

- Vererbung
  - http://www.cplusplus.com/doc/tutorial/inheritance/
- Virtuelle Methoden / abstrakte Klassen / Polymorphie
  - http://www.cplusplus.com/doc/tutorial/polymorphism/
- Zeitmessung
  - man 3 clock
  - man 3 gettimeofday