Programmieren in C++

SS 2012

Vorlesung 11, Dienstag 17. Juli 2012

(Type Casting, Vererbung & Konstruktoren, Projekt)

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 10. Übungsblatt (Vererbung)
  - Dies ist die **vorletzte** Vorlesung … Evaluationsbogen
- Vererbung II
  - Außer public und private gibt es noch: protected
  - Vererbung von Konstruktoren und sogenannte Initializer
  - static_cast, dynamic_cast, reinterpret_cast
- Und dann endlich …
  - Das **PROJEKT**
  - Übungsblatt 11: Entscheiden Sie sich für ein Projekt und schreiben Sie alle .h Dateien dazu (vorher nachdenken)

Erfahrungen mit dem Ü10 (Vererbung)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 17. Juli 13:31
  - Wieder (viel) mehr Arbeit als die letzten Übungsblätter
  - Aber nicht wegen Vererbung, sondern wegen der Geometrie
  - Von daher etwas unglückliche Aufgabenstellung
  - Das Dreieck haben einige weg gelassen
  - Die Tests waren aufwändig aber wenig hilfreich hier
  - Allgemeiner Endsemesterstress war wohl ein Teil des Problems
  - Gab aber auch viele, die keine Probleme hatten
  - Warum kein protected und Tiefergehendes erklärt? ... heute
  - Bildschirmgröße auf Jenkins mal 0, mal 65536, mal ...
  - Warum funktioniert pow(x, 1/3) nicht, aber pow(x, 0.333) ?

Offizielle Evaluation der Vorlesung

UNI FREIBURG

- Bitte den Bogen **bis Ende der Woche** abgeben
  - Ich würde das Feedback dann nämlich gerne in der letzten Vorlesung zusammenfassen und besprechen
  - Sie bekommen dafür 20 Punkte!
  - Diese Punkte ersetzen dann die Punktezahl von Ihrem schlechtesten Übungsblatt
  - Schreiben Sie dazu einfach in Ihre erfahrungen.txt, dass Sie den Evaluationsbogen ausgefüllt haben (wenn es so ist)
  - Nehmen Sie sich bitte **genug Zeit** für das Ausfüllen
  - Die Freitextkommentare sind für uns am interessantesten
  - Seien Sie bitte ehrlich und möglichst konkret

protected 1/2

UNI FREIBURG

- Alternative zu public oder private
  - In der Klasse wo es steht, selbe Wirkung wie private
  - Insbesondere brauch man FRIEND_TEST wenn man im Test Zugriff auf protected Membervariablen will
  - Bei der Vererbung werden Membervariable die private sind allerdings nicht vererbt, die protected schon

```
class AnimatedObject {
  public:
    void animationStep();  // Will be inherited by subclass.
  protected:
    size_t _objectSize;  // Will also be inherited.
  private:
    std::string debugInfo;  // Will not be inherited.
};
```

protected 2/2

UNI FREIBURG

- Was bei der erbenden Klasse steht ist auch wichtig

```
class AnimatedTriangle : public AnimatedObject {
  // What is public / protected in AnimatedObject
  // ... will also be public / protected in AnimatedTriangle.
};
```

  - Es setzt die **maximalen** Zugriffsrechte in der Unterklasse

```
class AnimatedTriangle : protected AnimatedObject {
  // Both public and protected from AnimatedObject
  // ... will now only be protected in AnimatedTriangle.
};
```

```
class AnimatedTriangle : private AnimatedObject {
  // Both public and protected from AnimatedObject
  // ... will now only be private in AnimatedTriangle.
};
```

Konstruktoren 1/5

UNI FREIBURG

- Implizite Aufrufshierarchie
  - Ohne Weiteres ruft **jeder** Konstruktor einer Unterklasse ganz am Anfang erstmal den **Default-Konstruktor** (= der Konstruktor ohne Argument) der Oberklasse auf
    - Der wiederum ruft dann, bei mehrstufiger Vererbung, den Konstruktor seiner Oberklasse auf, und so weiter ...
  - Dasselbe dann bei den Destruktoren, nur in umgekehrter Reihenfolge
  - Siehe folgendes Beispiel

Konstruktoren 2/5

UNI FREIBURG

- Implizite Aufrufshierarchie, Beispiel

```
class AnimatedObject {
  AnimatedObject() { _size = 10; }
  AnimatedObject(int size) { _size = size); }
  int _size;
};

...

class AnimatedTriangle : public AnimatedObject {
  AnimatedTriangle() { }
};

...

AnimatedTriangle triangle;
printf("%d\n", triangle._size);  // Prints 10.
```

Konstruktoren 3/5

UNI FREIBURG

- Explizite Aufrufe
  - Möchte man einen **anderen** Konstruktor aufrufen, gibt man das in der **Initialisierungsliste** des Konstruktors an
  - Der Defaultkonstruktor wird dann **nicht** mehr aufgerufen

```
class AnimatedTriangle : public AnimatedObject {
  explicit AnimatedTriangle(int size)
    : AnimatedObject(size) {
    ... // Random code.
  }
};

...

AnimatedTriangle triangle(20);
printf("%d\n", triangle._size);  // Prints 20.
```

Konstruktoren 4/5

UNI FREIBURG

- Explizite Aufrufe
  - Den Konstruktor aus dem Code aufrufen geht in C++ **nicht**

```
void AnimatedTriangle::someMethod() {
  ...
  // The following lines will not call the constructor for
  // this object, but will create temporary objects instead.
  AnimatedTriangle(20);
  AnimatedObject(10);
  ...
}
```

  - Falls der Konstruktor Code enthält, den man anderer Stelle gerne nochmal ausführen möchte:
  - Code in eine separate Methode, z.B. reset() oder initialize()

Konstruktoren 5/5

UNI FREIBURG

- Initialisierung von Membervariablen
  - Allgemeiner kann man in der erwähnten Initialisierungsliste auch **beliebige Membervariablen** initialisieren
  - Wobei man das besser einfach in den Code schreibt

```
class AnimatedObject {
  public:
    AnimatedObject(int size, int color)
      : _size(size), _color(color) {  // Initializer list.
      _size = size;  // But you can set it here just as well.
    };
  private:
    int _size;
    int _color;
};
```

Type Casting 1/6

UNI FREIBURG

- Implizite Konvertierung
  - Wir haben schon gesehen, dass Zeiger auf die Unterklasse bei Bedarf automatisch in Zeiger auf die Oberklasse konvertiert werden

    ```
    Thing* thing = new IntegerThing(4);  // No problem.
    ```

  - Andersrum aber nicht

    ```
    Thing* thing = new IntegerThing(4);
    ...
    IntegerThing* integerThing = thing;  // Will not compile.
    ```

Type Casting 2/6

UNI FREIBURG

- Explizite Konvertierung mit **dynamic_cast**
  - Man kann den Compiler aber dazu zwingen

```
Thing* t = new IntegerThing(4);
IntegerThing* it  = dynamic_cast<IntegerThing*>(t);  // Ok.
```

  - Bei dynamic_cast schaut das Programm **zur Laufzeit**, ob der Zeiger auf ein Objekt vom richtigen Typ zeigt
    - falls ja, wird die Konvertierung durchgeführt
    - falls nein, wird NULL zurückgegeben

```
Thing* t = new CharacterThing(4);
IntegerThing* it  = dynamic_cast<IntegerThing*>(t);
printf("%p\n", it);  // Will print NULL.
```

  - Für ein realistisches Beispiel wo man das braucht, siehe http://ad-svn...cplusplus-ss2010...vorlesung-11/Number.{h, cpp}

Type Casting 3/6

UNI FREIBURG

- Explizite Konvertierung mit **static_cast**
  - Das schreibt man völlig analog

```
Thing* t = new IntegerThing(4);
IntegerThing* it  = static_cast<IntegerThing*>(t);  // Ok.
```

  - Das prüft allerdings **nicht** nach, ob das sinnvoll ist, sondern kopiert einfach die Zeiger-Adresse
  - Den static_cast benutzt man auch für explizite Umwandlungen zwischen Basistypen

```
double pi = 3.141592653589793;
int piRounded1 = pi;  // Implicit conversion, will compile.
int piRounded2 = static_cast<int>(pi);  // Explicit is better.
size_t noPos1 = -1;  // Again, will compile.
size_t noPos2 = static_cast<size_t>(-1);  // But better this.
```

Type Casting 4/6

UNI FREIBURG

- Explizite Konvertierung mit **static_cast**
  - Mit static_cast kann man auch Konstruktoren mit einem Argument aufrufen, wo explicit davor steht

```
explicit ArrayInt(int x);  // Create array with one element.

...

// Method that appends array to the array object calling it.
void ArrayInt::append(const ArrayInt& array) { ... }

...

ArrayInt array;
array.append(5);  // Will not compile, because of explicit above.
array.append(static_cast<ArrayInt>(5));  // That will compile.
```

Type Casting 5/6

UNI FREIBURG

- Explizite Konvertierung mit **reinterpret_cast**
  - Bei static_cast müssen die Klassen immerhin verwandt sein

```
class Xyz {   };  // Class unrelated to Thing;

Thing* t = new IntegerThing(4);
Xyz* xyz  = static_cast<Xyz*>(t);  // Will not compile.
```

  - Mit reinterpret_cast kann man **beliebige** Zeiger-konvertierungen machen, und auch Zeiger ↔ Integer

```
Thing* t = new IntegerThing(4);
Xyz* xyz  = reinterpret_cast<Xyz*>(t);  // Will compile.
size_t x = reinterpret_cast<size_t>(t);  // Will compile.
```

  - Das braucht man allerdings in der Praxis nur ganz selten, bei (sehr) low-level Anwendungen, z.B. Treibersoftware

Type Casting 6/6

UNI FREIBURG

- Wir merken uns
  - Konvertierung SubClass* nach SuperClass* geht **automatisch** und braucht man praktisch immer sobald man was mit Vererbung macht
  - Konvertierung andersrum geht mit dynamic_cast und dann auf NULL checken, braucht man aber selten
    - Wenn man das braucht, Klassendesign hinterfragen!
  - Implizite Typkonvertierungen mit static_cast ausdrücklich hinschreiben ist eine gute Idee und kommt häufiger vor
  - Oder bei explicit Konstruktoren, das ist wieder eher selten
  - Und reinterpret_cast brauchen wir praktisch nicht

Projekt

UNI FREIBURG

- Sie haben 2 bzw. 3 Projekte zur Auswahl
  - Projekt 1: Snake
    - Eine ASCII Version des gleichnamigen Video-Spiels
  - Projekt 2: Virenscanner
    - Untersucht gegebene Datei nach Virensignaturen
  - Projekt 3: Selbstdefiniert
    - Für die, die in C++ schon sehr sicher / erfahren sind, und für die die Übungsblätter (zu) leicht waren
  - Details dazu auf dem Wiki ... insbesondere zu geforderter und optionaler Funktionalität, und einige technische Tipps

Projekt

UNI FREIBURG

- Übungsblatt 11
  - Schreiben Sie **alle** .h Dateien **aller** Klassen, die Sie zur Realisierung Ihres Projektes brauchen
  - Sie brauchen noch **nichts** implementieren, wirklich nur die .h Dateien ... eventuell den einen oder anderen Test
  - Aber schauen Sie, dass die .h Dateien alle kompilieren (indem Sie alle in einer ...Main.cpp includen, die sonst nichts weiter tut) und checkstyle ohne Fehler passieren
  - Nächste Woche bekommen Sie dann Feedback von Ihren Tutoren zu Ihren .h Dateien
  - Außerdem heute schon mal etwas Hintergrundwissen und Tipps zur Implementierung

Projekt 1 — Snake 1/2

UNI FREIBURG

- Prinzip
  - ASCII Animationen hatten wir ja jetzt schon öfter
  - Wir haben das bisher mit ANSI escape codes gemacht
  - Die sind einfach, aber auch etwas obskur

```
printf("\x1b[2J");  // Clears the screen
printf("\x1b[1;1H");  // Put cursor in upper left corner.
printf("\x1b[?25l");  // Hide the cursor
```

  - Es gibt auch eine Bibliothek dazu, curses bzw. ncurses, die ist sehr mächtig und außerdem portabler als die ANSI codes

```
man 3 ncurses
```

  - Beispielcode in vorlesung-12/GetKeyMain.cpp vom SS 20**11**

Projekt 1 — Snake 2/2

UNI FREIBURG

- ANSI escape codes vs. ncurses
  - Für einfache Sachen sind die ANSI escape codes prima, weil man keinerlei Funktionsaufrufe hat, sondern einfach nur (einfache) Kontrollsequenzen auf den Bildschirm druckt
  - Wenn man, bei der Erfassung von Tastendrücken, verhindern will, dass die entsprechenden Zeichen auf dem Bildschirm erscheinen, wird es ohne ncurses aber schon tricky

```
struct termios term = {0};
tcgetattr(0, &term);
term.c_lflag &= ~ICANON & ~ECHO;
tcsetattr(0, TCSANOW, &term);
```

  - Das geht mit ncurses einfacher und portabler

```
cbreak();
```

Projekt 2 — Virenscanner 1/3

UNI FREIBURG

Hex = Zahl zur Basis 16 schreiben.. Ziffern sind: 0, 1, ..., 9, a, b, c, d, e, f

eine Hex-Ziffer steht also für 4 Bits = 1 Nibble

- Grundprinzip eines Virenscanners
  - Einfache Viren sind durch eine Abfolge von Bytes charakterisiert, die irgendwo im Code stehen, z.B.

    c7b601b10132c060cd2672059d6142ebf6cd20

    Das sind 19 Bytes von Trojan.DiskEraser.20 in Hex-Code
  - Auf dem Wiki finden Sie eine Textdatei mit den Signaturen von über 30.000 Viren
  - Bei etwas komplizierteren Viren braucht man Signaturen mit sogenannten Wildcards, z.B. bei Worm.Hybris.D-2

    10400081??????????81??04000000??75f1*104000c3
  - Dabei steht ? für ein beliebiges Nibble = ein **halbes** Byte
  - Und * steht für eine beliebige Folge von Bytes

Projekt 2 — Virenscanner 2/3

UNI FREIBURG

- Fortsetzung + ein paar Tipps zur Umsetzung
  - Dann gibt es noch sowas wie

    ...{40-130}... → zwischen 40 und 130 beliebige Bytes

    ...{-5}... → bis zu 5 beliebige Bytes

    ...{80-}... → mindestens 80 beliebige Bytes

    ...{10}... → genau 10 beliebige Bytes
  - Die Umsetzung von ? und sowas wie {10} ist einfach
    - einfach so viele Nibbles / Bytes überspringen, und dann schauen ob der Rest der Signatur passt
  - Bei * oder {...-...} ist es schwieriger, weil man nicht weiß, nach wie vielen Bytes der Rest der Signatur weiter geht

Projekt 2 — Virenscanner 3/3

UNI FREIBURG

- Lücken variabler Größe
  - Nehmen wir zum Beispiel

    64???c40a466{12-20}e664b8*8ec026
  - Dann suchen wir erstmal **alle** Vorkommen der Stücke

    64???c40a466 und e664b8 und 8ec026

    Beachte: die Stücke können **mehrmals** vorkommen
  - Dann schauen wir ob es Kombinationen von diesen Vorkommen im passenden Abstand zueinander gibt
  - Noch ein paar Beispiele dazu auf dem Wiki
  - Diese Projekt ist also eher was für die, die gerne ein bisschen algorithmisch denken

Literatur / Links

- Type Casting
  - http://www.cplusplus.com/doc/tutorial/typecasting/
- Konstruktoren
  - http://www.cplusplus.com/doc/tutorial/classes/
  - http://www.cplusplus.com/doc/tutorial/inheritance/
- Projekte
  - Spezifikation auf dem Wiki zur Veranstaltung

    http://ad-wiki...ProgrammierenCplusplusSS2012/Projekt
  - Curses library

    http://heather.cs.ucdavis.edu/~matloff/.../Curses.pdf
  - ClamAV signature documentation

    http://www.clamav.net/doc/latest/signatures.pdf