Programmieren in C++

SS 2018

Vorlesung 5, Dienstag 29. Mai 2018

(Klassen und Objekte)

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 Ü4 — Game of Life
  - Aus gegebenem Anlass — Was ist ein Plagiat?
- Inhalt
  - Klassen und Objekte — Einführung + Beispielcode
  - Konstruktor und Destruktor — Einführung + Beispielcode
  - **Übungsblatt 5:** Game of Life objekt-orientiert machen

Erfahrungen mit dem Ü4 1/3

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Game of Life sehr interessant und schön anzuschauen, wenn es am Ende funktioniert hat
  - "Viel rumprobiert für eine funktionierende Lösung"
  - "An sich nicht schwer, aber bei C++ kann man seine Lebenszeit scheinbar an den kleinsten Dingen verlieren"
  - const int MAX und **nicht** extern const int MAX

    Siehe Forum (Topic 4578 "Deklarieren eines Arrays")
  - Habe zu oft Game of Life gespielt um zu wissen, wie lange ich für die eigentliche Aufgabe gebraucht habe
  - Schrödingers Terminal entdeckt: sobald man sich Zahlenwerte ausgeben lässt, verschwindet der Bug

Erfahrungen mit dem Ü4 2/3

UNI FREIBURG

- Sind wir Figuren in einem zellulären Automaten?
  - Durch bloße Anwesenheit wird niemand schwanger
  - Habe Gott gefragt, der meinte ich existiere wirklich
  - Ich eher nicht, aber ein Kumpel hat wirklich drei Eltern
  - Hängt davon ab, ob es wirklichen Zufall gibt oder nur scheinbaren: dann befänden wir uns in einem Automaten
  - Wirklichkeit ist zufällig, Game of Life ist deterministisch
  - Ja, sind wir: eine Woche nichts für die Uni zu machen war sowohl vorhersagbar, als auch reproduzierbar
  - Nein, sonst gäbe es in der Vorlesung größere Dynamik zwischen aufwachenden und einschlafenden Studenten

Erfahrungen mit dem Ü4 3/3

UNI FREIBURG

- Sind wir Figuren in einem zellulären Automaten?
  - Game of Life ist **Turing vollständig** = alles was man überhaupt berechnen kann, geht auch in Game Of Life
  - In der Tat kann man im Game Of Life zahlreiche (teil-weise sehr) komplexe Strukturen bauen, zum Beispiel:

    Gliderfabriken, die Raumschiffe bauen

    Eine Art Terminal mit Pixeln

    Tetris

    Eine vollständige CPU mit ALU, RAM, etc
  - Also warum nicht auch komplexe Lebewesen mit Selbst-bewusstsein?

Plagiate 1/4

UNI FREIBURG

- Das 10. Gebot
  - Nochmal zur Erinnerung der Text von Seite 3 von Übungsblatt 1, der sinngemäß auch auf Folie 4 von Vorlesung 1 steht:

    Sie können gerne zusammen über die Übungsblätter nachdenken.

    Aber der Code bzw. die Lösungen müssen zu **100%** selber geschrieben werden.

    Auch das teilweise Übernehmen gilt als Täuschungsversuch, mit den entsprechenden Konsequenzen.

Plagiate 2/4

UNI FREIBURG

- Ein Beispiel

Code Person 1

```
vx /= length;
vy /= length;
usleep(5000);
if (time > 0) {
  time--;
} else {time = 150;}
```

Code Person 2

```
vx /= length;
vy /= length;
usleep(5000);
if (time > 0) {
  time--;
} else {time = 150;}
```

Wurde hier nur zusammen nachgedacht?

Oder voneinander Code übernommen?

Plagiate 3/4

- Noch ein Beispiel

Code Person 1

```
maxX = 155;
maxY = 40;
if (x >= maxX) {
  vx = vx   * (-0.9);
  x = maxX;
}
```

Code Person 2

```
maximalX = 145;
maximalY = 30;
if (x >= maximalX) {
  vDirX = vDirX   * (-0.9);
  x = maximalX;
}
```

Wurde hier nur zusammen nachgedacht?

Oder voneinander Code übernommen?

Plagiate 4/4

UNI FREIBURG

- Noch ein ganz anderes Beispiel
  - Was denken Sie, wie oft die folgenden Satzteile auf Google vorkommen? (mit Phrasensuche "...")

    "Das können Sie machen wo und wann Sie wollen"

    Anzahl Treffer: 1

    Und zwar in einer Kopie meiner Vorlesungsfolien vom SS 2013 auf docplayer.org

    "Übernehmen gilt als Täuschungsversuch"

    Anzahl Treffer: 0

    Das ist von den Folien von der ersten Vorlesung vom SS 2018, die noch nicht von Google indiziert wurde

Klassen 1/13

UNI FREIBURG

- Sinn einer Klasse
  - Ganz grob gesagt: in einer Klasse wird Code zusammen-gefasst, der zusammengehört
  - Wie bei unserem bisherigen Code auch, unterscheiden wir zwischen Variablen und Funktionen
  - In einer Klasse nennt man die **Membervariablen** und **Memberfunktionen** (Member = Mitglied)
  - Die Memberfunktionen nennt man auch **Methoden**
  - Wie bei Variablen und Funktionen, unterscheidet man auch bei Klassen zwischen Deklaration und Implementierung

Klassen 2/13

UNI FREIBURG

- Deklaration einer Klasse
  - Das macht man einfach wie folgt:

```
class GameOfLife {
   …  // Contents of the class, see next slides.
};
```

  - Das schreibt man in eine **.h** Datei, die man genau so nennt wie die Klasse, in dem Fall GameOfLife.h

    In Java ist das ein Muss, in C/C++ nicht zwingend, es erhöht aber sehr die Lesbarkeit des Codes

    In Java **kein** Semikolon am Ende der Deklaration

    Wenn man das Semikolon in C/C++ vergisst, gibt es mitunter sehr merkwürdige Fehlermeldungen

Klassen 3/13

UNI FREIBURG

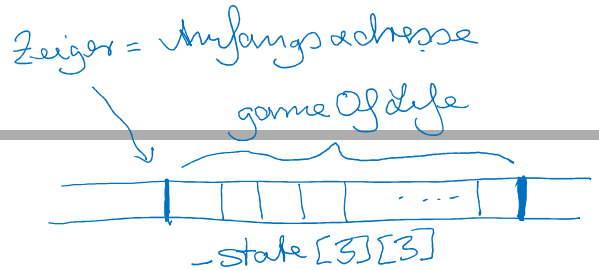
- Membervariablen einer Klasse
  - Die Membervariablen schreibt man wie bei einer normalen Deklaration, nur **in** die Klasse (in der .h Datei)

```
class GameOfLife {
   bool _state[200][200];   // The current state.
};
```

  - Beachte: der **Unterstrich** am Beginn des Namens ist eine Konvention, um die Membervariablen leicht von normalen Variablen unterscheiden zu können
  - Es können **beliebig viele** Membervariablen in einer Klasse stehen

    Für GameOfLife vom Ü5 braucht man auch mehrere

Klassen 4/13



- Objekte einer Klasse
  - Im Code kann man jetzt Objekte dieser Klasse erzeugen:

    GameOfLife gameOfLife;

    Das geht also im Prinzip genauso wie bei der "normalen" Variablendeklaration: links der Typ, rechts der Name
  - **Wichtig:** wie bei der Deklaration einer normalen Variablen reserviert das bereits Speicherplatz, und gameOfLife ist der Name für dieses Stück Speicher

    In **Java** würde die gleiche Anweisung nur eine **Referenz** (= einen Zeiger) auf ein Objekt von diesem Typ erzeugen

    In C++ gibt es auch Zeiger auf Objekte, das brauchen wir aber erstmal nicht → spätere Vorlesung

Klassen 5/13

UNI FREIBURG

- Methoden einer Klasse, Deklaration
  - Die **Methoden** (Memberfunktionen) werden wie die Membervariablem **in** der Klasse deklariert:

```
class GameOfLife {
  bool _state[200][200];
  void updateState();  // Update the current state.
};
```

  - Beachte: da die Methode zur Klasse gehört, gibt es keinen Grund, sie gameOfLifeUpdateState() o.ä. zu nennen
  - Es gibt auch keinen Grund für einen **Unterstrich** vor dem Namen der Funktion, das macht man nur bei den Membervariablen

Klassen 6/13

UNI FREIBURG

- Methoden einer Klasse, Implementierung
  - Wie bei normalen Funktionen auch, steht die Implementierung in einer gleichnamigen **.cpp** Datei

```
void GameOfLife::updateState() {
   …;  // Code that updates the current state.
}
```

  - Beachte: der Klassenname ist jetzt Teil des Namens der Funktion und **muss** hier mit angegeben werden

    In Java gibt es auch solche **voll qualifizierten** Methodennamen, die würde dann aber in dem Fall hier GameOfLife**.**updateState() heißen, also mit **.** statt **::**

Klassen 7/13

UNI FREIBURG

- Zugriff auf Membervariablen und Methoden
  - Das geht über den **.** Operator

```
GameOfLife game;  // Create new object (short name ok).
for (int x = 0; x < 200; x++)
   for (int y = 0; y < 200; y++)
      game._state[x][y] = (x + y) % 3;
game.updateState();
```

  - **Wichtig:** die Zuweisung verändert die Werte von der Variablen _state von genau nur diesem Objekt

    Gäbe es ein anderes Objekt **game2**, so hätte das auch eine Variable **_state**, dessen Werte mit denen von **game** oben nichts zu tun haben

Klassen 8/13

UNI FREIBURG

- Zugriffsberechtigung
  - Dazu gibt es in der Klassendeklaration zwei Abschnitte

```
class GameOfLife {
  public:          // Note: indent of one space is enough
    ...
  private:         // Dito.
    ...
};
```

  - Auf alles, was nach public: steht, kann man wie auf den Folien vorher über den **.** Operator zugreifen
  - Auf alles, was nach private: steht, dürfen nur die Implementierungen der Methoden zugreifen

Es gibt auch noch protected: … siehe spätere VL

Klassen 9/13

UNI FREIBURG

- Zugriffsberechtigung, Beispiel
  - Die Methode **play()** sollte man **public** machen

    Grund: man will ja außerhalb der Implementierung der Klasse so was wie game.play() schreiben können
  - Die Variable **_state** sollte man **private** machen

    Grund: nur die Methoden der Klasse selber müssen den Inhalt der Zellen kennen, außerhalbe der Klasse reicht es, wenn man game.play() schreiben kann
  - Das ist gerade ein wichtiges Prinzip von objekt-orientierter Programmierung: man gibt nur so viel "nach außen" wie nötig, die Implementierung "versteckt" man

Klassen 10/13

UNI FREIBURG

- Statische Membervariablen
  - Das sind die (quasi "globalen") Variablen einer Klasse, die für alle Objekte der Klasse den gleichen Wert haben
  - Die deklariert man dann in der .h Datei so:

```
class SomeOtherClass {
   static int _numberOfObjects;
};
```

  - Die Initialisierung erfolgt dann in der .cpp Datei so:

```
int GameOfLife::_numberOfObjects = 0;
```

    Das keyword **static** steht nur bei der Deklaration !
  - Für das Ü5 brauchen Sie das erstmal nicht

Klassen 11/13

UNI FREIBURG

- **Friend** Klassen und Methoden
  - Manchmal möchte man Variablen private machen, aber ausgewählten anderen Klassen trotzdem Zugriff geben:

```
class Glider {
   friend class GameOfLife;  // Give access to coordinates.
   int _coordinates[3][3];
};

GameOfLife::processUserInput { …
   Glider glider;
   // Code can access glider._coordinates
}
```

  - Das geht auch nur für einzelne Methoden, siehe Referenzen

Klassen 12/13

UNI FREIBURG

- Vorwärtsdeklaration (brauchen wir noch nicht)
  - Manchmal hat man zwei Klassen X und Y, die müssen sich gegenseitig kennen
  - Das löst man mit einer **Vorwärtsdeklaration**

```
class Y;  // Forward declaration

class X {
   Y y;  // We can now use type Y in declarations.
};

class Y {
   X x;  // No problem anyway, since X declared before.
};
```

Klassen 13/13

UNI FREIBURG

- FRIEND_TEST
  - Zum **Testen** braucht man oft Zugriff auf alle Member-variablen einer Klasse … auch die, die private sind
  - Dazu schreiben wir in die Deklaration

```
#include <gtest/gtest.h>  // FRIEND_TEST defined here.
…
class GameOfLife {
   void updateState();
   FRIEND_TEST(GameOfLifeTest, updateState);
};
```

In der Test Datei muss dann entsprechend stehen:

```
TEST(GameOfLifeTest, updateState) { … }
```

Konstruktor und Destruktor 1/7

UNI FREIBURG

- Motivation
  - Objekte sind in C++ erstmal **nicht** initialisiert
  - Um Sie bei der Deklaration zu initialisieren braucht man einen Konstruktor

```
class GameOfLife {
   GameOfLife(bool fillValue);
  …
};
```

Heißt genauso wie die Klasse, aber ohne Rückgabewert

Kann auch Argumente haben und es kann auch mehrere Konstruktoren mit verschiedenen Argumenten geben

Bei genau einem Argument **explicit** davor schreiben → V6

Konstruktor und Destruktor 2/7

UNI FREIBURG

- Implementierung und Aufruf
  - Implementierung in der .cpp Datei

```
GameOfLife::GameOfLife(bool fillValue) {
   for (...) { _state[x][y] = fillValue; }
}
```

  - Aufruf erfolgt automatisch bei der Objekt-Deklaration

```
GameOfLife game(0);  // Calls constructor with argument 0.
```

    Wenn es mehrere Konstruktoren gibt, sucht sich der Compiler anhand der Argumente den passenden

  - Achtung: das mit dem bool fillValue wurde hier nur zur Demonstration benutzt, braucht man nicht für Ü5

Konstruktor und Destruktor 3/7

UNI FREIBURG

- Defaultkonstruktor
  - Wenn man keinen Konstruktur definiert, gibt es immer einen sog. **Defaultkonstruktor**, der (fast) nichts macht

    GameOfLife game; // Calls the **default constructor**.
  - Kann man überschreiben, in dem man einen Konstruktur **ohne Argumente** deklariert und implementiert

    class GameOfLife { GameOfLife(); … }; // In the .h file.

    GameOfLife::GameOfLife() { …} // In the .cpp file.
  - **Beachte:** Sobald man einen Konstruktor mit Argumenten deklariert, gibt es den Defaultkonstruktur nicht mehr

    Muss man dann bei Bedarf explizit selber schreiben, s.o.

Konstruktor und Destruktor 4/7

UNI FREIBURG

- Welcher Code gehört in den Konstruktor
  - Vor allem **einfache Variablenzuweisungen**
  - Komplexerer Code gehört in eine separate Methode, z.B.

    GameOfLife::initialize(); // Ncurses initialization.

    Die sollte dann auch separat im Code aufgerufen werden, und nicht im Konstruktor, das kann sonst zu gemeinen Fehlern führen, insbesondere bei globalen Objekten

    Details sind im Google C++ Styleguide beschrieben
  - Außerdem hat man so den Vorteil, dass man die initialize() Methode bei Bedarf wiederholt aufrufen kann

    Den Konstruktor kann man in C++ nicht explizit aufrufen

Konstruktor und Destruktor 5/7

UNI FREIBURG

- Destruktor, Motivation
  - Der **Destruktor** wird aufgerufen, wenn das Objekt seinen sogenannten "scope" verlässt.

    Der "scope" einer Variablen oder eines Objektes beginnt bei der letzten **{** davor und endet an der dazugehörigen **}**

```
void someFunction() {
   // A new scope has just begun.
   ...
   GameOfLife game;  // Constructor called.
   ...
   // Scope ends, destructor of "game" will be called.
}
```

Konstruktor und Destruktor 6/7

UNI FREIBURG

- Destruktor, Deklaration und Implementierung
  - Der Destruktor heißt wie der Konstruktur, nur mit **einer Tilde** davor

```
class GameOfLife {                    // In the .h file
   ~GameOfLife();
};

GameOfLife::~GameOfLife() {     // In the .cpp file.
    endwin();
}
```

  - Üblicherweise räumt der Destruktor auf, was immer es aufzuräumen gibt ... im Beispiel oben den Bildschirm

Konstruktor und Destruktor 7/7

UNI FREIBURG

- Defaultdestruktor
  - Wenn man explizit keinen Destruktor scheibt, gibt es immer einen impliziten Defaultdestruktor, der nichts macht

    Anders als beim Konstruktor gibt es aber nur einen Destruktor ... insbesondere gibt es keine Destruktoren mit Argumenten

Literatur / Links

UNI FREIBURG

- Alles zu Klassen
  - http://www.cplusplus.com/doc/tutorial/classes/
- Friend Klassen und Methoden
  - http://www.cplusplus.com/doc/tutorial/inheritance/

    Abschnitte "Friend functions" und "Friend Classes"
  - http://code.google.com/p/googletest/wiki/AdvancedGuide

    Abschnitt "Testing Private Code"
- Game Of Life Computer
  - https://codegolf.stackexchange.com/questions/11880/build-a-working-game-of-tetris-in-conways-game-of-life