Programmieren in C++

SS 2022

Vorlesung 5, Dienstag 24. Mai 2022

(Klassen und Objekte)

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 Ü4 — Game of Life
  - Plagiate — Aus gegebenem Anlass
- Inhalt
  - Wir machen heute zusammen eine vereinfachte Version von "Snake" zusammen objektorientiert
  - Klassen und Objekte — Einführung + Beispielcode
  - Konstruktor und Destruktor — Einführung + Beispielcode

Ü5: Game of Life objektorientiert machen

Wir bereiten dazu heute wieder sehr viel zusammen vor

Erfahrungen mit dem Ü4 1/3

UNI FREIBURG

- Feedback zur Aufgabe
  - Die Aufgabe hat wieder den meisten gefallen
  - Für die meisten einfacher und besser machbar als das Ü3

    Gründe: Viel gelernt aus den Fehlern vom Ü3, Vertrautheit mit ncurses, nützliche Vorgaben in der Vorlesung
  - Einige hätten sich mehr Erklärung zu Zeigern gewünscht

    Problem: Zeiger sind konzeptionell sehr einfach (es sind einfach Adressen), aber den Nutzen begreift man erst mit der Zeit, durch die eigene Anwendung und Erfahrung
  - Einige haben sich den Tipp, "inkrementell" zu coden, zu Herzen genommen und davon profitiert

Erfahrungen mit dem Ü4 2/3

UNI FREIBURG

- Ausgewählte Zitate

  "Vorlesung war gut und die Übung hat voll Spaß gemacht"

  "Inkrementell ist super und Testen auch, leider erst spät auf den Geschmack gekommen"

  "In kleineren Stücken gearbeitet, was [sehr] geholfen hat"

  "Routine kommt langsam rein"

  "Existentielle Krise durch das GameOfLife entwickelt"

  "Ich finde im Gegensatz zu E.i.d.P sind die Übungsblätter spannend und machen Spaß"

  "Wäre geil, wenn mehr Testfunktionen eingeführt werden"

  "Mit Zeigern hadere ich noch"

  "Achte vermehrt darauf, keinen vierten Nachbarn zu bekommen"

Erfahrungen mit dem Ü4 3/3

UNI FREIBURG

- Einführung in die Programmierung
  - Ca. 80% haben die Veranstaltung gehört
  - Bei einigen schon länger her und es wurde öfter angemerkt, dass man schnell wieder aus der Übung kommt
  - Den meisten hat E.i.d.P. gut gefallen, sie sind gut mitgekommen und haben auch die Übungsblätter bearbeitet
  - Bei den übrigen 20% diverse Programmiervorkenntnisse
  - Fazit: Probleme bei der "Einführung in die Programmierung" waren kein wesentlicher Grund für die Probleme beim Ü3

    Gut zu wissen, vielen Dank für die Rückmeldungen

Plagiate 1/4

UNI FREIBURG

- Das 10. Gebot
  - Aus gegebenem Anlass nochmal zur Erinnerung:

    Sie können gerne zusammen über die Übungsblätter nachdenken

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

    Auch das teilweise Übernehmen von Code (von anderen, aus dem Internet, aus vorherigem Semester) gilt als Täuschungsversuch

    Einzige Ausnahme: Code aus /public oder /loesungen aus dem SVN von diesem Semester

Plagiate 2/4

- 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

- Bitte halten Sie sich an die Regeln!
  - Wir sind ziemlich gut im Auffinden von Plagiaten

    Auch beim Ü3 sind wir leider mehrmals fündig geworden, teils mit Verschleierungsversuch
  - So gut zu plagiieren, dass wir es nicht merken, ist ungefähr so schwierig, wie das Übungsblatt richtig zu bearbeiten

    Für die, die es betrifft: bitte nehmen Sie sich das zu Herzen und halten Sie sich an die Regeln!

Klassen 1/11

UNI FREIBURG

- Sinn einer Klasse
  - Ganz grob gesagt: In einer Klasse wird Code zusammen-gefasst, der zusammengehört

    Das erhöht auch die Wiederverwendbarkeit von Code, siehe die Klasse **TerminalManager**, die wir heute zusammen coden
  - 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 Definition (Implementierung)

Klassen 2/11

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

    Das Semikolon nach der } nicht vergessen

Klassen 3/11

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 grid_[200][200] = { 0 };
   int numSteps_ = 0;
};
```

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

Klassen 4/11

UNI FREIBURG

- 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

Klassen 5/11

UNI FREIBURG

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

```
class GameOfLife {
  bool grid_[200][200] = { 0 };
  int numSteps_ = 0;
  void play();
};
```

  - Beachte: da die Methode zur Klasse gehört, gibt es keinen Grund, sie playGameOfLife () o.ä. zu nennen
  - Es gibt auch keinen Grund für einen **Unterstrich** nach dem Namen der Funktion (da es in der Regel keine "lokalen Funktionen" gibt, mit denen man sie verwechseln könnte)

Klassen 6/11

UNI FREIBURG

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

```
void GameOfLife::play() {
   …;  // Code for playing (used to be in "main").
}
```

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

Klassen 7/11

UNI FREIBURG

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

    ```
    GameOfLife gameOfLife;
    gameOfLife.updateState();
    ASSERT_EQ(gameOfLife.numSteps_, 1));
    ```

  - **Wichtig:** die Zuweisung verändert die Werte von der Variablen numSteps_ von genau nur diesem Objekt; gäbe es ein zweites Objekt von dem Typ, hätte das auch eine Variable numSteps_ , deren Wert mit dem oben nichts zu tun hat

    ```
    GameOfLife gameOfLife2;
    gameOfLife2.updateState();
    ASSERT_EQ(gameOfLife2.numSteps_, 1));
    ```

Klassen 8/11

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 9/11

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 **numSteps_** 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 10/11

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) { … }
```

Klassen 11/11

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 GameOfLife {
   static const int MAX_SIZE = 200;
};
```

Das brauchen Sie für das Ü5, mehr zu const in V6

  - Es geht auch static ohne const, aber das braucht man nur selten (zum Beispiel static int numObjects_ = 0, zum Zählen der Anzahl der Objekte einer Klasse)

Konstruktor und Destruktor 1/7

UNI FREIBURG

- Motivation
  - Ein Objekt muss man typischerweise **initialisieren**, bevor man es sinnvoll (oder überhaupt) benutzen kann
  - Man könnte dafür eine Methode initialize schreiben, aber dann darf man nicht vergessen, die aufzurufen
  - Deshalb gibt es in C++ sogenannte **Konstruktoren**, die automatisch bei der Definition des Objektes aufgerufen werden

    ```
    GameOfLife gameOfLife(40, 80);
    ```

    Ein Konstruktor ist wie eine Funktion und kann insbesondere Argumente haben, der Compiler ermittelt anhand der Argumente, welcher Konstruktor aufgerufen wird

Konstruktor und Destruktor 2/7

UNI FREIBURG

- Deklaration und Definition
  - Deklaration in der .h Datei; der Konstruktor heißt **genau** so, wie die Klasse, nur ohne Rückgabetype

```
class GameOfLife {
  GameOfLife(int numRows, int numCols);  // Constructor.
  ...
};
```

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

  - Definition (Implementierung) in der .cpp Datei

```
GameOfLife::GameOfLife(int numRows, int numCols) {
   numRows_ = numRows;
   numCols_ = numCols;
}
```

Konstruktor und Destruktor 3/7

UNI FREIBURG

- Initialisierung von Membervariablen im Konstruktor
  - Membervariablen im Konstruktor lassen sich auch wie folgt initialisieren

```
GameOfLife::GameOfLife(int numRows, int numCols)
  : numRows_(numRows), numCols_(numCols) {
   …;  // Other code.
}
```

  - Vorteil 1: Falls in der .h Datei nach numRows_ und numCols_ Variablen deklariert sind, die von numRows_ und numCols_ abhängen, werden diese initialisiert wie erwartet (siehe SVN)
  - Vorteil 2: Falls die Objekte nach dem : keinen Basistyp haben, wird der passende Konstruktor aufgerufen → siehe V6

Konstruktor und Destruktor 4/7

UNI FREIBURG

- Defaultkonstruktor
  - Der Defaultkonstruktor ist der Konstruktor **ohne Argumente**
  - Den gibt es in C++ auch ohne, dass man ihn explizit definiert

    Dieser implizite Defaultkonstruktor initialisiert jede Membervariable entsprechend ihrer Definition in der .h Datei
  - Den impliziten Defaultkonstruktor kann man überschreiben, indem man explizit einen Konstruktor ohne Argumente schreibt

    `class TerminalManager { TerminalManager(); … };` .h Datei

    `TerminalManager::TerminalManager() { …}` .cpp Datei
  - **Beachte:** Sobald man einen Konstruktor mit Argumenten deklariert, gibt es den impliziten Defaultkonstruktor nicht mehr

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 TerminalManager {                                    .h Datei
   ~TerminalManager();
};
```

```
TerminalManager::~TerminalManager() {                      .cpp Datei
    endwin();
}
```

  - Üblicherweise steht im Destruktor Code, der das aufräumt, was im Konstruktor aufgebaut (oder während der Lebenszeit des Objektes) wurde

    Im Beispiel oben wird der Bildschirm wieder "zurückgesetzt"

Konstruktor und Destruktor 7/7

UNI FREIBURG

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

    Für Membervariablen, die Basistypen sind (z.B. int oder bool[100]) tut der nichts

    Für Membervariablen, die keine Basistypen sind, wird (rekursiv) deren Destruktor aufgerufen, in umgekehrter Reihenfolge, in der sie in der .h Datei stehen

    Ein explizit definierter Destruktor tut das auch, nachdem er den Code des Destruktors ausgeführt hat
  - Anders als beim Konstruktor gibt es nur einen Destruktor

    Insbesondere gibt es keinen Destruktor mit Argumenten

Literatur / Links

- 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"