Programmieren in C++

SS 2012

Vorlesung 5, Dienstag 5. Juni 2012

(Klassen, Objekte, Methoden, new & delete)

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 4. Übungsblatt
- Themen heute
  - Erste objektorientierte Schritte
  - Dynamische Speicherallokation mit new und delete
  - Übungsblatt dazu:
    - Code vom letzten Übungsblatt objektorientiert machen
    - … und mit dynamischer Speicherallokation
  - Anmerkung: wenn Sie Probleme mit dem letzten Übungsblatt hatten, können Sie einfach die Musterlösung nehmen und damit weiter machen

Erfahrungen mit dem Ü4 (Wurm)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 5.6 13:30
  - Aufwändigstes / schwierigstes Übungsblatt bisher
  - Aber angemessen + hat Spaß gemacht
  - Diesmal weniger Vorgaben / Hilfestellung
  - Für viele dadurch interessanter, für einige dadurch sehr schwer
  - Unit-Tests haben einige Fehler gefunden
  - gdb hat bei einigen segmentation faults geholfen
  - Problem mit aufeinander aufbauenden Übungsblättern?
  - Fehler beim Rotieren des Richtungsvektors … separate Folie
  - Normalisierung des Richtungsvektors … war nicht gefordert
  - Musterlösung vom Makefile besprechen … andermal
  - Kann ich Zwischenversionen ins SVN machen? … Ja klar!

Rotation des Richtungsvektors

UNI FREIBURG

- Ein klassischer (und nicht leicht zu findender) Fehler
  - "Update" von einem Vektor, bei uns etwa:

```
dirX = dirX – dirY;
dirY = dirX + dirY;
```

  - Fehler: in der zweiten Zeile hat dirX schon den neuen Wert!
  - Einfache Lösung dafür:

```
int newDirX = dirX – dirY;
int newDirY = dirX + dirY;
dirX = newDirX;
dirY = newDirY;
```

  - Bei sowas helfen einem Unit Tests insofern, dass Sie einem zumindest sofort zeigen, **wo** im Code der Fehler liegt
  - Was dann der Fehler ist ... nachdenken (manchmal nötig)

Klassen

UNI FREIBURG

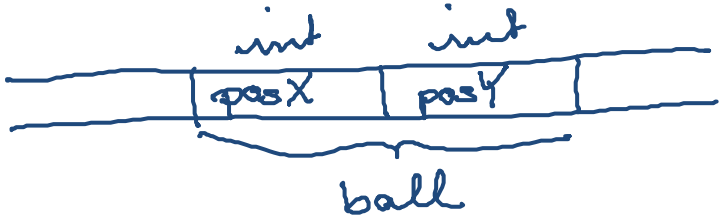
- Für uns erstmal zwei Verwendungszwecke
  - Definition von neuen (komplexeren) Datentypen
    - durch die **Membervariablen** einer Klasse
  - Zusammenfassen von Funktionen die "zusammen gehören"
    - das sind die **Methoden** einer Klasse

Membervariablen einer Klasse

UNI FREIBURG

- Deklaration der Klasse
  - Man schreibt einfach alle Variablen, die zu dem Datentyp gehören sollen in die Klasse, wie bei einer Deklaration

```
class Ball {
   int posX;
   int posY;
};
```



- Deklaration eines Objektes der Klasse
  - Analog zur Deklaration für die Basisdatentypen:

```
Ball ball;
```

  - **Wichtig:** Das reserviert bereits Speicherplatz, und ball ist der Name für dieses Stück Speicher
  - In **Java** wäre das nur eine Referenz (= ein Zeiger)

Methoden eine Klasse

UNI FREIBURG

- Zu der Klasse gehörige Funktionen heißen **Methoden**
  - Sie werden innerhalb der Klasse (in der .h Datei) deklariert

```
class Ball {
  void move();
  int posX;
  int posY;
};
```

  - Implementierung in der .cpp Datei; man beachte, dass zum vollen Methodennamen der Klassenname gehört

```
void Ball::move() {
  posX++;
  posY++;
}
```

Zugriff auf Membervariablen & Methoden

UNI FREIBURG

- Das geht über den **.** Operator

```
Ball ball;  // Create object of class Ball.
printf("Position: (%d,%d)\n", ball.posX, ball.posY);
ball.move();
```

  - Die letzte Zeile ruft die move() Funktion auf
  - Dieser Aufruf ändert posX und posY von genau (und nur) dem Objekt ball, auf dem sie aufgerufen wird

- Zugriffsberechtigung
  - Damit das kompiliert, braucht man **Zugriffsberechtigung** für die entsprechenden Membervariablen / Methoden → nächste Folie

Zugriffsberechtigung

UNI FREIBURG

- Public und Private
  - Vor den Abschnitt mit den Membervariablen und Methoden, auf die von außen (über ein Objekt) zugegriffen werden soll schreibt man (**nur ein** Leerzeichen eingerückt)

    public:
  - Vor den Abschnitt mit den Membervariablen und Methoden, auf die von außen (über ein Objekt) **nicht** zugegriffen werden soll, schreibt man (**nur ein** Leerzeichen eingerückt)

    private:
  - Es gibt auch noch protected ... später (Vererbung)
  - In den Methoden einer Klasse kann man grundsätzlich auf alle Membervariablen und Methoden dieser Klasse zugreifen
    - auch von anderen Objekten der Klasse ... nächste VL

Benennung von Membervariablen 1/2

UNI FREIBURG

- Wie unterscheidet man Membervariablen von normalen Variablen?

```
void Ball::move() {
  int posX = posX + 1; // Local variable with same name.
  printf("Value is: %d\n, posX);  // What is printed now?
  ...
};
```

  - Bei Mehrdeutigkeiten nimmt der Compiler die **zuletzt** deklarierte Variable
  - Aber besser man vermeidet solche Mehrdeutigkeiten!
  - Dazu versehen wir alle Membervariablen mit einem _ (Unterstrich bzw. englisch Underscore)

Benennung von Membervariablen 2/2

UNI FREIBURG

- Das sieht dann so aus:

```
class Ball {
  void move();
  int _posX;
  int _posY;
};
void Ball::move() {
  _posX++;
  _posY++;
}
```

  - Das ist nur eine Namenskonvention, sonst nichts!
  - Aber eine, die hilft, doofe Fehler zu vermeiden

Konstruktor 1/4

UNI FREIBURG

- Objekte sind ohne Weiteres erstmal **un**initialisiert
  - Anders als bei den Basistypen (int, char, double, bool, etc.) ist es allerdings üblich, sie zu initialisieren
  - Das tut man über den sogenannten Konstruktor
  - Der Konstruktor ist eine Methode, die so heißt wie die Klasse und keinen Rückgabewert hat
  - Deklaration innerhalb der Klasse in der .h Datei

```
class Ball {
  Ball(int posX, int posY);
  int _posX;
  int _posY;
};
```

Konstruktor 2/4

UNI FREIBURG

- Implementierung und Aufruf
  - Implementierung in der .cpp Datei

```
Ball::Ball(int posX, int posY) {
  _posX = posX;
  _posY = posY;
}
```

  - Aufruf erfolgt automatisch bei der Objekt-Deklaration

```
Ball ball(10, 10);  // Calls the constructor.
printf("Position = (%d,%d)\n", ball._posX, ball._posY);
```

  - Die letzte Zeile druckt dann Position = (10,10)

    (vorausgesetzt _posX und _posY sind public)

Konstruktor 3/4

UNI FREIBURG

- Defaultkonstruktor
  - So heißt der Konstruktor ohne Argumente

    Ball ball; // Calls the **default constructor**.
  - Wenn es sonst keinen Konstruktur gibt (und nur dann), gibt es den Defaultkonstruktur ohne dass man etwas tut
    - er tut dann ... nichts
  - Man kann ihn aber auch explizit in der .h Datei deklarieren

    Ball();

    ... und in der .cpp Datei implementieren

```
Ball::Ball() {
  _posX = 1;
  _posY = 1;
}
```

Konstruktor 4/4

UNI FREIBURG

- Komplexerer Code gehört **nicht** in den Konstruktur
  - Sondern in eine separate Methode, z.B. initialize()
  - Das kann sonst zu schwer zu findenden Fehler führen
  - Insbesondere bei: globalen Objekten, Vererbung, Exceptions
  - Für Details, siehe

    http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Doing_Work_in_Constructors

FRIEND_TEST

UNI FREIBURG

- Membervariablen sind in der Regel private
  - Weil der Zugriff auf die Werte eines Objektes meist über die Methoden der Klasse ist
    - Die Idee dahinter ist, dass ein Benutzer der Klasse nicht wissen muss, wie das Objekt innen aussieht
  - In den Tests will man aber oft Zugriff auf die private Member-variablen, z.B. im TEST(BallTest, move)

```
Ball ball;
ball.setPosition(1, 1);
ASSERT_EQ(1, ball._posX);  // Compiler error: posX is private.
```

  - Dazu schreibt man in der Klasse **unter** die Methodendeklaration

```
move();
FRIEND_TEST(BallTest, move);
```

Statische Membervariablen

UNI FREIBURG

- Variablen, die zu der Klasse als Ganzes gehören

  … und nicht zu jedem einzelnen Objekt

  – Zum Beispiel so etwas wie _maxX und _maxY

  – Die deklariert man dann in der Klasse so:

```
static int _maxX;
static int _maxY;
```

  – Bei **nicht const** Variablen muss die Initialisierung in der .cpp Datei erfolgen (separat von der Deklaration)

```
int Ball::_maxX = 20;
int Ball::_maxY = 100;
```

Dynamische Speicherallokation 1/3

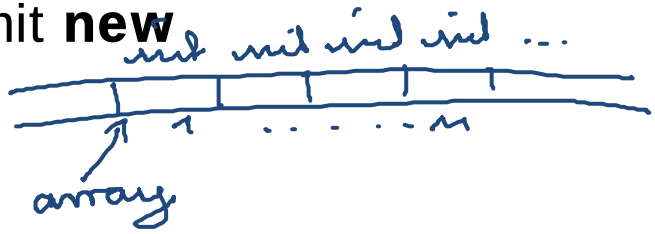
UNI FREIBURG

- Statische Speicherallokation
  - Bei der Deklaration von einem Feld muss die Größe des Feldes bekannt sein

    ```
    int n = atoi(argv[1]);
    int array[n];  // This will not compile, n must be const.
    ```

- Dynamische Speicherallokation
  - Zur Laufzeit bekommt man Speicher mit **new**

    ```
    int n = atoi(argv[1]);
    int* array = new int[n];
    ```

  - Das **new** alloziert in dem Fall Speicherplatz für n ints und gibt einen Zeiger auf den Anfang dieses Platzes zurück

Dynamische Speicherallokation 2/3

UNI FREIBURG

- Dynamische Allokation von Objekten
  - Geht analog

    ```
    int* p1 = new int;  // Allocate space for a single int.
    ```

  - Für Klassen wird dabei der Konstruktor aufgerufen

    ```
    Ball* p2 = new Ball(10, 10);
    ```

  - Ohne Argumente der Defaultkonstruktor (DK)

    ```
    Ball* p3 = new Ball();  // Will not compile without the ().
    ```

  - Bei Feldern von Objekten wird für jedes der DK aufgerufen

    ```
    Ball* p4 = new Ball[10];  // Will create 10 Ball objects.
    ```

  - Anmerkung: in **Java** kann man nur so (mit new) konkrete Objekte erzeugen; man muss sich anders als in **C++** aber nicht um das Aufräumen kümmern ... nächste Folie

Dynamische Speicherallokation 3/3

UNI FREIBURG

- Freigeben von Speicher
  - Dynamisch allozierten Speicher sollte man freigeben wenn man ihn nicht mehr braucht, das geht mit **delete**

```
int n = atoi(argv[1]);
int* array = new int[n];  // Pointer to an array of n ints.
int *p1 = new int;  // Pointer to single int.
int* p2 = new Ball();  // Pointer to single object.
... some code ...
delete[] array;  // For pointers to arrays use delete[].
delete p1;  // For pointers to var's of basic types use delete.
delete p2;  // Dito for pointers to single objects.
```

Destruktor 1/2

UNI FREIBURG

- 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.
   ...
   Ball ball;  // Constructor called.
   ...
   // Scope ends now, destructor of "ball" will be called.
}
```

Destruktor 2/2

UNI FREIBURG

- Der Destruktor ist auch einfach eine Methode
  - Und heißt wie der Konstruktur, nur mit einer ~ davor
  - In der Deklaration der Klasse in der .h Datei:

    ```
    ~Worm();
    ```

  - Implementierung in der .cpp Datei:

    ```
    Worm::~Worm() {
      delete[] _posX;  // Assuming we had _posX = new int[...] before.
      delete[] _posY;  // Dito.
    }
    ```

  - Wie beim Konstruktor muss man nicht explizit einen Destruktor deklarieren / implementieren
  - Es wird dann am Lebensende des Objektes der **Default-Destruktor** aufgerufen ... der einfach nichts tut

Literatur / Links

UNI FREIBURG

- Klassen, Objekte, Methoden, Konstr. und Destr.
  - http://www.cplusplus.com/doc/tutorial/classes/
- Dynamische Speicherallokation mit new & delete
  - http://www.cplusplus.com/doc/tutorial/dynamic/
- FRIEND_TEST
  - http://code.google.com/p/googletest/wiki/AdvancedGuide#Testing_Private_Code