Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

27. September 2019

Klassen

Inhalt

- Enum-Klassen
- Deklaration/Implementation
- Beispielklasse `Point`
- Keyword `const`
- Erzeugen statischer/dynamischer Instanzen
- `this`-Pointer
- Konstruktoren & Destruktoren
- RAII
- spezielle Konstruktoren
- Default-Parameter
- Default-Methoden

Enum-Klassen

- Stark-typisierter Enum
- Automatischer Scope für Konstanten
- Der darunter-liegende Typ kann definiert werden
- Default-Typ entspricht erster Konstante

```
1  enum class Color : uint8_t {
2    kRed = 1,
3    kGreen = 2,
4    kBlue = 4,
5  };
6
7  Color c = Color::kBlue;
8
9  // abfragen des unterliegenden Typs
10 using EnumType = std::underlying_type<Color>::type;
```

Deklaration/Implementation

- Header-Datei für Deklaration
  - deklariert die Klasse und ihre Attribute, Konstruktoren, Methoden, Operatoren, etc.
  - inkludiert andere benötigte Interfaces
- Source-Datei für Definition/Implementation
  - inkludiert zugehörigen Header
  - implementiert die deklarierten Methoden
  - definiert Memory-Space für statische Variablen
  - inkludiert intern benötigte Interfaces

point.h

```
#include "color.h"  // wird in Interface verwendet

class Point {
 public:
  void Print() const;
  // inline getter
  Color GetColor() const { return m_color; }

 private:
  double x_;
  double y_;
  double z_;
  Color color_;
};
```

point.cpp

- Implementierung der Methode `Print()` aus der Klasse `Point`
- die Signatur muss der vorgegebenen in der Klasse entsprechen

```
1  #include "point.h"       // dazuehörige Interface definition
2
3  #include <iostream>      // wird nur intern verwendet
4
5  void Point::Print() const {
6    std::cout << "Point ("
7      << x_ << "," << y_ << "," << z_
8      << ") with Color: " << GetColor() << std::endl;
9  }
```

Verwendung von `Point`

```
1  #include "point.h"
2
3  int main() {
4    {
5      Point p;     // Instanz auf dem Stack
6      p.Print();
7    }              // Punktobjekt p wird hier zerstört
8    // p.Print(); // Compile-Error
9
10   return EXIT_SUCCESS;
11 }
```

Was wird hier ausgegeben?

Nicht klar, da un-initialisierte Variablen in `Point`

Verwendung von `Point`

```
1  #include "point.h"
2
3  int main() {
4    Point* pp = nullptr;  // Pointer auf dem Stack
5    {
6      pp = new Point();   // Instanz auf dem Heap
7      pp->Print();
8    }
9    pp->Print();          // erlaubt
10
11   delete pp;            // dynamisches Punktobjekt zerstören
12   pp = nullptr;
13   return EXIT_SUCCESS;
14 }
```

Keyword const

```
1  struct Line {
2    Line(double x, double y, double z) : begin_(x, y, z)
3    {} // zwingend in Initialisierungs-Liste
4
5    // darf keinen Member modifizieren -> transitiv
6    Point GetEnd() const { return end_ };
7
8    // doppelt nicht erlaubt
9    Point GetBegin() const { begin_ = end_; return begin_; };
10
11  private:
12   const Point begin_;
13   Point end_;
14 };
```

Erzeugen einer statischen Instanz

Syntax: `<classname> <variable>;`

Beispiel: `Point p;`

Was passiert im Hintergrund?

- Compiler hat aus der Definition der Klasse `Point` berechnet, wie viel Speicher eine Instanz der Klasse benötigt
- in unserem Beispiel sind drei Attribute vom Typ `double` und ein Attribut vom Typ `Color` bzw. `uint8_t` vorhanden
- Speicherbedarf pro Instanz: `3 * sizeof(double) + 1 * sizeof(uint8_t)`
- auf dem Stack wird entsprechend Platz reserviert, so dass alle Attribute der Instanz abgespeichert werden können
- die Variable `p` bezeichnet die Instanz auf dem Stack

Erzeugen einer dynamischen Instanz

Syntax: `<classname>* <variable> = new <classname>();`

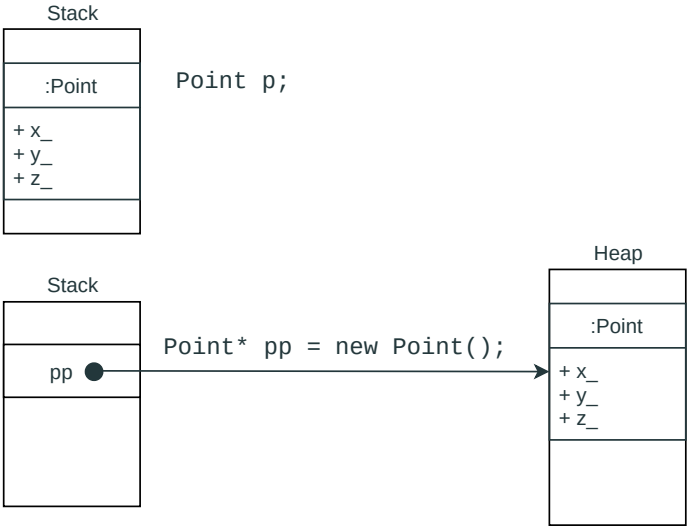
Beispiel: `Point* p = new Point();`

- statische und dynamische Variablen
  - `p` ist eine statische Variable vom Typ „Pointer auf `Point`“
  - `p` zeigt auf eine anonyme Instanz der Klasse `Point`
  - mit `delete` wird eine dynamische Instanz auf dem Heap zerstört und abgeräumt

Was passiert im Hintergrund?

- auf dem Heap wird genügend Platz für eine neue Instanz der Klasse alloziert und dieser Speicherbereich wird mit Aufruf des Konstruktors teilweise initialisiert
- die Speicheradresse (ein Pointer auf die neue Instanz) wird zurückgegeben und in der Pointervariablen `p` abgespeichert

Erzeugen statischer/dynamischer Instanzen



Anonyme (temporäre) Instanzen

- Instanzen ohne Namen
- werden nur kurzfristig benutzt
- werden nach Aufruf automatisch zerstört

```
1  // anonyme Instanz
2  Point(1, 2, 3).Print();
3  // wird zerstört
4
5  // schlecht: verwaiste dynamische Instanz
6  (new Point(2, 3, 4))->Print();
7  // lebt weiter auf Heap -> Memory Leak
```

Klassen-Variablen und -Methoden

- werden pro Klasse und nicht pro Instanz angelegt
- Keyword: `static` und Zugriff über `<classname>::`

```
1  struct Point {
2    Point();
3    ~Point();
4
5    static int GetCount();
6
7   private:
8    double x_, y_, z_;
9
10   static int count_;
11 };
```

```
1  int Point::count_ = 0;
2
3  Point::Point()
4      : x_(0), y_(0), z_(0) {
5    count_++;
6  }
7  Point::~Point() {
8    count_--;
9  }
10
11 int Point::GetCount() {
12   return count_;
13 }
```

this-Pointer

- zeigt auf die eigene Instanz
- kann in Instanz-Methoden verwendet werden

```
1  Point& Move(double delta[3]) {
2    x_ += delta[0];
3    y_ += delta[1];
4    z_ += delta[2];
5    return *this;
6  }
7
8  const double delta[] = {1, 2, 3};
9  Point p(0, 0, 0);
10 p.Move(delta).Move(delta);      // method chaining
11 p.Print()                      // Position: (2, 4, 6)
```

Konstruktoren

- primitive Datentypen besitzen keine Konstruktoren
  - und müssen manuell initialisiert werden
- Konstruktoren heissen gleich wie Klasse
  - und haben keinen return-Wert
- können nur bei Instanzierung aufgerufen werden
  - nicht zur Re-Initialisierung

```
1  class Point {
2   public:
3    // Default-ctor
4    Point() : x_(0), y_(0), z_(0), color_(Color::kBlue) {}
5    // benutzerdefinierter ctor
6    Point(double x, double y, double z, Color color)
7      : x_(x), y_(y), z_(z), color_(color) {}
8  }
```

Destruktoren

- Destruktor: gleiche Signatur plus ~ (Tilde)
- wird durch `delete` oder bei out-of-scope auf Stack aufgerufen
- leerer Standard-dtor wenn nicht selber definiert

```
1  class Point {
2   public:
3    ~Point() {
4      ...
5      std::cout << "destruction finished" << std::endl;
6    }
7  }
```

Initialisierungs- und Zerstörungsreihenfolge

- ctor initialisiert Attribute in Deklarations-Reihenfolge
  - danach folgt eigener Rumpf
- dtor zerstört Attribute in umgekehrter Reihenfolge
  - nachdem eigener Rumpf ausgeführt wurde

```
1  struct A { ... };
2  struct B { ... };
3  struct C {
4    A a_;
5  };
6  struct D {
7    A a_;
8    B b_;
9    C c_;
10 };
```

```
1  {
2    D instance;
3      // ctor A
4      // ctor B
5      // ctor A
6      // ctor C
7      // ctor D
8
9  }   // Zerstörung
```

RAII - Resource Allocation is Initialization

- einer der Gründe weshalb es Destruktoren gibt
- nach ctor muss Instanz korrekt initialisiert sein
- nach dtor müssen alle Resourcen freigegeben worden sein

```
1  class FileHandler {
2   public:
3    FileHandler(const std::string& path) {
4      file_ = fopen(path.c_str(), "r");
5    }
6    ~FileHandler() {
7      fclose(file_);
8    }
9
10   private:
11    FILE file_;
12 };
```

RAII - Error Handling

- wenn in ctor etwas schief geht:
  - Aufräumen und
  - Exception werfen
- dtor wird nicht aufgerufen
  - aber alle internen bereits erzeugten Objekte werden zerstört

```
1  class FileHandler {
2   public:
3    FileHandler(const std::string& path) {
4      file_ = fopen(path.c_str(),"r");
5        if (file_ == nullptr) {
6          throw std::invalid_argument("file not found");
7        }
8    }
```

Default-Parameter

- Parameter in Funktionen dürfen Default-Werte haben
  - diese werden nur im Header angegeben
- für Default-Parameter dürfen, müssen aber keine Werte beim Aufruf angegeben werden
- zuerst alle Parameter ohne Default-Wert
- dürfen in Funktionen, Methoden und Konstruktoren eingesetzt werden

```
1  // Farbe ist standardmässig auf Grün gesetzt
2  Point(double x, double y, double z, Color color = Color::kGreen)
3      : x_(x), y_(y), z_(z)
4      , color_(color)
5  {}
```

Kopier-Konstruktor/-Operator

- kopieren von Instanzen (flache oder tiefe Kopie)
- genau einen Parameter in Form einer const-Ref auf dieselbe Klasse
- üblicherweise sollte auch Assignment-Operator definiert werden

```
1  Point(const Point& other)
2      : x_(other.x_), y_(other.y_), z_(other.z_)
3      , color_(other.color_)
4  {}
5
6  Point& operator=(const Point& other) {
7    x_ = other.x_;
8    y_ = other.y_;
9    z_ = other.z_;
10   color_ = other.color_;
11   return *this;
12 }
```

Typkonvertierungs-Konstruktoren

- genau einen Pflicht-Parameter als const-Ref auf einen anderen Typ
- funktioniert auch wenn alle anderen Parameter Default-Werte haben
- kann zur impliziten Konvertierung verwendet werden
- meist will man nur explizite Konvertierungen für eigene Klassen

```
1  explicit Point(double pos[3], const Color& color = Color::kGreen)
2      : x_(pos[0]), y_(pos[1]), z_(pos[2])
3      , color_(color)
4  {}
5
6  double value[] = {1, 2, 3}
7  Point p(value);
8  p = value;            // Compile-Error
```

Default-Methoden

- Compiler synthetisiert automatisch diverse Methoden wenn nicht explizit definiert
- diese Synthetisierung kann manuell beeinflusst werden mit `delete` oder `default`

Synthetisierte Methoden

ctor	nur wenn kein user-defined ctor
dtor	nur wenn kein user-defined dtor
copy	nur wenn kein user-defined move
move	nur wenn kein user-defined copy & dtor & move für alle Attribute gültig ist

Beispiel: Default-Methoden

```
1  class Point {
2    explicit Point(double pos[3], const Color& color = Color::kGreen)
3        : x_(pos[0]), y_(pos[1]), z_(pos[2])
4        , color_(color)
5    {}
6
7    // automatisch generiertes Copy
8    Point(const Point& other) = default;
9    Point& operator=(const Point& other) = default;
10
11   // explizites Verhindern von Move-Semantik
12   Point(Point&& other) = delete;
13   Point& operator=(Point&& other) = delete;
```