Programmiersprachen
Sommersemester 2019

Inhaltsverzeichnis

1 Style Guide & Best Practice

1.1 Header-Dateien

Verwenden Sie Header-Dateien (`*.hpp`) für eine saubere Trennung von Deklaration und Definition von Funktionen/Methoden.

Üblicherweise existiert zu jeder `*.cpp`-Datei eine zugehörige `*.hpp`-Datei. Eine Ausnahme bildet die Datei, die die `main()`-Funktion enthält.

1.1.1 Include Guards

Alle Header besitzen ein mit **`#define`** gesetztes Symbol um Mehrfachinkludierung zu vermeiden, den sogenannten *Include Guard*. Das Format dieses Symbols sollte `<PROJECT>_<DATEI>_HPP` sein (Quelltext 1).

```
#ifndef RAYTRACER_SPHERE_HPP
#define RAYTRACER_SPHERE_HPP

class Sphere {
  ... // Klassendefinition
};

#endif // RAYTRACER_SPHERE_HPP
```

Quelltext 1: Verwendung von Include Guards

Hinweis: Vermeiden Sie die Nutzung von **`#pragma once`** anstelle der expliziten Include-Guards! Die **`#pragma once`**-Direktive ist in keinem C++-Standard definiert und behindert die Erzeugung portablen Codes.[1]

1.2 `#include`-Direktive

Die Präprozessor-Direktive **`#include`** ist ein Befehl für den Präprozessor, der den Inhalt der referenzierten Datei anstelle des **`#include`** ersetzt.

1.2.1 Relative vs. absolute Include-Pfade

Der Präprozessor-Direktive **`#include`** folgt immer genau ein Dateipfad als Argument. Je nachdem, ob relative Pfade ausgehend vom Verzeichnis der

[1] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#sf8-use-include-guards-for-all-h-files

Quellcode-Datei oder absolute bzw. Standard-Suchpfade benutzt werden sollen, wird der Pfad entweder mit Anführungszeichen oder spitzen Klammern markiert. Standard-Suchpfade sind Systemabhängig, aber für den Compiler nach Installation bekannt. Unter Linux-Systemen ist ein typischer Standard-Suchpfad z.B: `/usr/include`.
Als Beispiel seien die ersten Zeilen des Inhalts einer Datei namens example.cpp gezeigt.

```
// sucht Datei namens 'sphere.hpp'
// im gleichen Verzeichnis in dem diese Datei liegt
#include "sphere.hpp"

// sucht Datei namens 'vector' in allen
// Standard-Suchpfaden (z.B. /usr/include/c++/7/)
#include <vector>
```

Quelltext 2: Verwendung von lokalen und Standard-Include-Pfaden

1.2.2 Reihenfolge der Includes

Benutzen Sie in Ihren Programmierprojekten folgende Reihenfolge:

1. Projektzugehörige `*.hpp`-Dateien, (z.B. `sphere.hpp`)
2. C++-Fremdbibliotheken (z.B. `glm.hpp` oder `boost/filesystem.hpp`)
3. C++-Standardbibliotheken (z.B. `algorithm` oder `vector`)

Es folgt ein Beispiel mit der lokalen Datei `sphere.cpp`:

```
// Include von lokalem / projektinternem File
#include "sphere.hpp"

// Include von File aus einer Fremdbibliothek
#include <boost/filesystem.hpp>
#include <glm/glm.hpp>

// Standard-Headers
#include <algorithm>
#include <vector>


int main(int argc, char** argv) {
  // ...
  return 0;
}
```

Quelltext 3: Reihenfolge der Includes

1.3 Gültigkeitsbereich/Scoping

1.3.1 Lokale Variablen

Platzieren Sie die Variablen einer Funktion in den engstmöglichen Gültigkeitsbereich und initialisieren Sie diese zur Deklaration.

```
//Schlecht:
  // Wert von i unklar
  unsigned int i;
  //...
  // i wird erst hier gebraucht, nicht engstmoeglich
  for (i = 0; i < 10; ++i) {
    // ...
```

```
  }
  // 'i' existiert hier weiter.. Fehlergefahr!
//-----------------------------------------------

//Gut:
/* i wird bei der Initialisierung
   gezielt ein Wert zugewiesen  */

  for (unsigned int i = 0; i < 10; ++i) {
    // ...
  }
  // 'i' lebt nur im Gueltigkeitsbereich
  // der for-Schleife
```

1.3.2 Namespaces

Benutzen Sie **niemals** `using namespace X;` im globalen Scope. Explizite Verwendung von Namespaces verringert die Gefahr doppelter Namensvergebung bei Typen und Funktionen drastisch.

```
//Schlecht:
#include <cmath>
#include <iostream>

using namespace std;

int main() {
  float const pi = 3.14159265359f;
  cout << "sinus of pi:" << sin(pi) << endl;

  return 0;
}
//-----------------------------------------------

//Gut:
#include <cmath>
#include <iostream>

int main() {
  float const pi = 3.14159265359f;
  std::cout << "sinus of pi:"
```

```
        << std::sin(pi) << std::endl;

  return 0;
}
```

1.4 Klassen

1.4.1 Ressourcenbelegung ist Initialisierung

Alle für ein Objekt notwendigen Initialisierungen finden im Konstruktor statt. Dies wird häufig mit RAII bezeichnet (*resource acquisition is initialization*).

1.4.2 `struct` vs. `class`

Nutzen Sie ein `struct` nur für Datentransferobjekte; alles andere wird als `class` definiert. Klassen sollten **keine** public Member beinhalten. [2]

1.4.3 Initialisierungsliste

Initialisieren Sie alle Membervariablen in der Initialisierungsliste eines Konstruktors. Verwenden Sie dabei die gleiche Reihenfolge wie in der Klassendefinition.

```
//sphere.hpp
#ifndef RAYTRACER_SPHERE_HPP
#define RAYTRACER_SPHERE_HPP

#include "point2D.hpp"
#include "colorrgb.hpp"

#include <string>

class Sphere {
public:
  Sphere(std::string const& name,
         Point2D const& center, float radius,
         colorRGB const& color);
  // ...

private:
```

[2] http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c8-use-class-rather-than-struct-if-any-member-is-non-public

```
                std::string     name_;
                Point2D         center_;
                float           radius_;
                ColorRGB        color_;

#endif //RAYTRACER_SPHERE_HPP
```

Quelltext 4: Sphere.hpp mit Konstruktor und Membervariablen

```
//sphere.cpp
#include "sphere.hpp"

Sphere::
Sphere(std::string const& name, Point2D const& center,
       float radius, colorRGB const& color)
         : name_{name}, center_{center},
           radius_{radius}, color_{color}
         {} /* leerer Konstruktorrumpf,
               da alle Member initialisiert sind */
```

Quelltext 5: Sphere.cpp mit Konstruktor und Initialisierungsliste

1.5 Namen

Beschreibende Namen können dazu beitragen Code verständlich zu machen und die Menge an notwendigen Kommentaren zu vermeiden.

1.5.1 Generell

Verwenden Sie möglichst deskriptive Namen. Vermeiden Sie unleserliche Abkürzungen wie z.B `GenYmDhMs`.

1.5.2 Variablen

Variblennamen beginnem mit einem Kleinbuchstaben. Halten Sie die Namen kurz und einfach. Verwenden Sie für Indices zum Beispiel `i`, `j`, `k`. Ansonsten wählen Sie möglichst deskriptive Namen wie z.B. `node` für einen Knoten.

Vermeiden Sie unnötige Bezeichneranhänge wie `my_circle`.

Schreiben Sie Variablen, die aus mehr als einem Wort bestehen im *snake_case*. z.B. `root_node`.

Kennzeichnen Sie Membervariablen einer Klasse mit einem Unterstrich am Ende z.B.: `glm::vec3 color_`.

1.5.3 Typen/Klassen

Datentypen und Klassen beginnen mit einem Großbuchstaben. Verwenden Sie Substantive. Zusammengesetzte Namen schreiben Sie im *CamelCase*.

```
class BoundingBox;
```

1.5.4 Methoden/Memberfunktionen

Verwenden Sie einfache englisch Verben oder kurze englische Ausdrücke. Funktions- und Methodennamen werden klein geschrieben. Zusammengesetzte Worte trennen Sie mit Unterstrichen (snake_case).

```
  BoundingBox Sphere::bounding_box() const {
    // definition von bounding_box() methode
  }
```

1.6 Formatierung

1.6.1 Leerzeichen vs. Tabs

Benutzen Sie nur Leerzeichen, keine Tabs. Rücken Sie ihren Code mit 2 Leerzeichen ein.

```
//Schlecht
#include <iostream>
int print_decimals()
{
for(unsigned i=0;i<10;++i){
std::cout << i << std::endl;
}
}
int main(int argc, char* argv[])
{
print_decimals();
return 0;
}
```

```
//Besser
#include <iostream>

int print_decimals()
{
  for(unsigned i=0; i < 10; ++i) {
    std::cout << i << std::endl;
  }
}

int main(int argc, char* argv[])
{
  print_decimals();
  return 0;
}
```

1.7 `const`-correctness

Der `const`-Qualifiers stellt je nach Anwendungskontext sicher, dass Zustände von Objekten nicht verändert werden. In Kombination mit der Frage nach const-correctness tritt gleichzeitig die Frage nach der Parameterrückgabe per-Value oder per-Referenz auf.
Wir unterscheiden zwischen 4 Anwendungsgebieten von `const` mit folgenden regeln:

1. `const`-Übergabeparameter
   - Versprechen, dass das übergebene Objekt nicht verändert wird
   - Übergabe von nicht-elementaren Datentypen: `const&`, wenn das übergebene Objekt nicht verändert werden soll
2. `const`-Rückgabewerte
   - `const` (per-Value): in der Regel nicht sinnvoll, da es nur verspricht, dass das temporäre Objekt nicht verändert werden darf
   - `const&` (per-Referenz): je nach Anwendungsfall können Rückgabetypen die per-Referenz zurückgegeben werden `const` oder nicht `const`-qualifiziert zurückgegeben werden
3. `const` für Konstanten
   - Werte, die sich über den Programmlauf nicht verändern sollen
4. `const`-Member-Funktionen

- das Methoden-`const` verspricht, dass die Methode den `Zustand` des Objekts, auf dem es Aufgerufen wird, nicht verändert

Durch die konsequente Benutzung des `const`-Qualifiers lassen sich viele potenzielle Bugs schon während der Compile-Zeit erkennen. Außerdem stellt

1.8 Diverses

Vermeidung von Code-Duplizierung

Nutzen Sie für Ihre Implementierung bereits vorhandene Funktionen. Implementieren Sie z.B. einen Operator für eine freie Funktion mittels eines kompatiblen Operators einer Memberfunktion. (z.B. `operator*` mittels `operator*=`)

Arrays

Benutzen Sie `std::array` anstelle der alten C-Arrays.

```
//Schlecht:
  int a[5];
//----------------------

//Gut:
  std::array<int,5> a;
/* Alternativ: (wenn Groesse erst zur
   Laufzeit bekannt ist) */
  std::vector<int> a(5);
```

Preincrement

Benutzen Sie die Prefixform (`++i`) für die Inkrementierung von Zählern und Iteratoren.
Neben den oben angegebenen Richtlinien erhalten Sie nachfolgend als Hilfestellung während des Programmierens und als Leitlinie für den Code den Sie zur Abnahme mitbringen sollten folgende Checkliste:

2 Checkliste für Ihren Code:

- Deskriptive Namen verwendet - **kein** `float my_radius = 0.0f;` oder ähnliches
- `const`-correctness einhalten - insbesondere `const&` bei Übergabe nicht-elementarer Datentypen und `const`-Memberfunktionen
- Signaturen: Übergabe- und Rückgabe-Qualifier (`const&`) korrekt nutzen
- Konzepte Datentransferobjekte (DTOs) vs. Klassen richtig einsetzen
  - DTOs:
    - haben keine explizit definierten Kosntruktoren, Destruktoren oder Memberfunktionen
    - Member werden durch Default Member Initialiser initialisert (z.B. `float pos_x = 0.0f;`)
    - Member-Variablem bei DTOs ohne Underscore am Ende (z.B. `float pos_x = 0.0f;`)
  - Klassen
    - **kein** direkter Zugriff auf Membervariablen möglich
    - Memberfunktionen benutzen Membervariablen (ansonsten sollten es freie Funktion sein)
    - **Keine** Java-Style Programmierung: **keine** Zugriffe auf Membervariablen mit `this->member_variable`
    - Member-Variablen bei Klassen mit Underscore am Ende (z.B. `float radius_;`)
- Konstruktoren: sind alle Member initialisiert? Member-Initialiser-List benutzt?
- Variablen im engstmöglichen Gültigkeitsbereich (Scope) definieren
- Variablen erst deklariert und initialisiert, wenn sie benötigt werden
- Verwendung von `auto` nur, wo es Lesbarkeit erhöht (nach Ermessen)
- Code sinnvoll nach obigen Richtlinien formatieren und benamen
  - CamelCase für Klassen und DTOs
  - snake_case für jegliche Art Variablen
  - snake_case für freie Funktionen und Methoden

Nichteinhaltung der vorgestellten Richtlinien führt zu Punktabzügen in den Belegabnahmen.