1 Freie Funktionen

Bei der Überladung von binären Operatoren wird die Variante der freien Funktion bevorzugt, da sie die implizite Konvertierung für beide Operanden ermöglicht. In dieser Aufgabe sollen Sie eine Klasse `OneSide` mit `operator==` als Member-Funktion und eine Klasse `TwoSide` mit freier Funktion implementieren.

1.1 Aufgabe

Implementieren Sie beide Klassen mit einem einzelnen **`int`** als Attribut und einem einzelnen Konstruktor, welcher dieses Attribut mit einem beliebigen Wert initialisiert. Damit dieser Konstruktor auch für implizite Konvertierungen von **`int`** zu `OneSide` resp. `TwoSide` funktioniert, dürfen Sie ihn nicht `explicit` markieren, stattdessen aber cpplint mittels `// NOLINT[runtime/explicit]` zufriedenstellen.

Nun können Sie den `operator==` einmal als Member-Methode (mit nur einem Parameter) in `OneSide` implementieren und einen entsprechenden Test schreiben. Sie werden feststellen, dass ein Ausdruck (`a` ist eine `OneSide` Instanz) wie **`if`** `(a == 1)` funktioniert, **`if`** `(1 == a)` aber nicht.

Implementieren Sie den `operator==` nun als freie Funktion (mit zwei Parameter und der `friend`-Markierung) in `TwoSide`. Mit dieser Variante sollten nun beide Variante des oben gezeigten Vergleichs funktionieren.

Versuchen Sie zu erklären, was hier genau passiert und weshalb die Variante mit Member-Methode nicht funktioniert.

1.2 Lösung

Bei dem erwähnten Ausdruck sucht der Compiler nach einem passenden `operator==` für die beiden Instanzen. Eine der Instanzen ist vom Typ `OneSide` oder `TwoSide` und die andere Instanz ist vom Typ **`int`**. Da er keinen passenden Operator findet, versucht der Compiler eine der beiden Instanzen mittels impliziter Konvertierung zum anderen Typ zu casten. Hier kommt nun unser Typkonvertierungs-Konstruktor zum Einsatz, welcher es erlaubt einen **`int`** zu einer unserer beiden Klassen zu casten.

Im Fall für `OneSide` findet er dadurch aber immer noch keinen passenden Operator, da er nur einen Parameter einer Funktion/Methode casten kann und nicht den Typ zu welchem die Methode gehört. Man kann sich das so vorstellen, dass in jeder Methode ein versteckter erster Parameter existiert, welcher den **`this`**-Pointer darstellt. Und genau diesen Parameter kann er nicht casten, da dieser ein Pointer-Typ ist.

freie_funktion_test.cpp

```cpp
#include <catch2/catch.hpp>

#include "one_side.h"
#include "two_side.h"

TEST_CASE("FreieFunktion Test", "[OneSide, TwoSide]") {
  SECTION("OneSide") {
    OneSide a = 1;

    REQUIRE(a == a);
    REQUIRE(a == 1);
    // REQUIRE(1 == a);  // funktioniert nicht mit operator== als Member
  }

  SECTION("TwoSide") {
    TwoSide a = 1;

    REQUIRE(a == a);
    REQUIRE(a == 1);
    REQUIRE(1 == a);     // funktioniert mit freier Methode
  }
}
```

one_side.h

```cpp
#pragma once

class OneSide {
 public:
  OneSide(int value);   // NOLINT[runtime/explicit]

  bool operator==(const OneSide& rhs) const;

 private:
  int value_;
};
```

one_side.cpp

```cpp
#include "one_side.h"

OneSide::OneSide(int value) : value_(value) {}

bool OneSide::operator==(const OneSide& rhs) const {
  return value_ == rhs.value_;
}
```

two_side.h

```h
#pragma once

class TwoSide {
 public:
  TwoSide(int value);   // NOLINT[runtime/explicit]

  friend bool operator==(const TwoSide& lhs, const TwoSide& rhs);

 private:
  int value_;
};
```

two_side.cpp

```cpp
#include "two_side.h"

TwoSide::TwoSide(int value) : value_(value) {}

bool operator==(const TwoSide& lhs, const TwoSide& rhs) {
  return lhs.value_ == rhs.value_;
}
```

2 Relationale Operatoren

Üblicherweise werden wenige bis keine Operatoren von eigenen Klassen definiert. Neben den Assignment und Stream-Operatoren sind aber die relationalen Operatoren häufig sinnvoll. Vorallem wenn die eigene Klasse innerhalb eines Containers wie z.B. `std::set` verwendet werden soll. Wir wollen uns hier somit Operator-Overloading zunutze machen.

Dabei sollte man sich zunutze machen, dass die meisten relationalen Operatoren durch einen der anderen definiert werden kann. Z.B. können Sie `!=` durch die Invertierung vom Resultat von `==` implementieren. Verwenden Sie auch hier freie Funktionen für die Implementierung der Operatoren.

Eine übersicht über die relationalen Operatoren finden z.B. hier: cppreference.com - Comparison operators. Der `operator<=>` wird hier bereits erwähnt, sollten Sie aber nicht verwenden, da der C++20-Standard noch nicht definitiv ist.

2.1 Aufgabe

Implementieren Sie die Klasse `ComparableData` welche ein **`int`** und ein **`double`** Attribut hat. Diese Klasse sollen Sie später in einer sortieren Menge vom Typ `std::set<ComparableData>` verwenden

können. Bei der Sortierung soll das `int` Attribut höhere Priorität haben. Schreiben Sie zudem ein paar Tests, welche die korrekte Sortier-Reihenfolge überprüfen.

2.2 Lösung

comparable_data.h

```
#pragma once

class ComparableData {
 public:
  ComparableData(int a, double b);

  friend bool operator==(const ComparableData& lhs, const ComparableData& rhs);
  friend bool operator<(const ComparableData& lhs, const ComparableData& rhs);

  friend inline bool operator!=(const ComparableData& lhs,
                                const ComparableData& rhs) {
    return !operator==(lhs, rhs);
  }

  friend inline bool operator>(const ComparableData& lhs,
                               const ComparableData& rhs) {
    return operator<(rhs, lhs);
  }

  friend inline bool operator<=(const ComparableData& lhs,
                                const ComparableData& rhs) {
    return !operator>(lhs, rhs);
  }

  friend inline bool operator>=(const ComparableData& lhs,
                                const ComparableData& rhs) {
    return !operator<(lhs, rhs);
  }

 private:
  int a_;
  double b_;
};
```

comparable_data.cpp

```
#include "comparable_data.h"

ComparableData::ComparableData(int a, double b) : a_(a), b_(b) {}

```

```cpp
bool operator==(const ComparableData& lhs, const ComparableData& rhs) {
  return lhs.a_ == rhs.a_ && lhs.b_ == rhs.b_;
}

bool operator<(const ComparableData& lhs, const ComparableData& rhs) {
  if (lhs.a_ < rhs.a_) {
    return true;
  } else if (lhs.a_ > rhs.a_) {
    return false;
  } else {
    return lhs.b_ < rhs.b_;
  }
}
```

comparable_data_test.cpp

```cpp
#include <catch2/catch.hpp>

#include <set>

#include "comparable_data.h"

TEST_CASE("ComparableData Test", "[OneSide, TwoSide]") {
  SECTION("Equals") {
    ComparableData a(1, 1);
    ComparableData b(1, 1);
    REQUIRE(a == a);
    REQUIRE(b == b);
    REQUIRE(a == b);
  }

  SECTION("Not Equals") {
    ComparableData a(1, 1);
    ComparableData b(2, 2);
    ComparableData c(1, 2);
    ComparableData d(2, 1);
    REQUIRE(a != b);
    REQUIRE(a != c);
    REQUIRE(a != d);
  }

  SECTION("Smaller") {
    ComparableData a(1, 1);
    ComparableData b(2, 2);
    ComparableData c(1, 2);
    ComparableData d(2, 1);
    REQUIRE(a < b);
    REQUIRE(a < c);
    REQUIRE(a < d);
```

```
34    }
35
36    SECTION("Bigger or Equals") {
37      ComparableData a(1, 1);
38      ComparableData b(2, 2);
39      REQUIRE(b >= b);
40      REQUIRE(b >= a);
41    }
42
43    SECTION("Sorted Set") {
44      std::set<ComparableData> set;
45      set.emplace(1, 2);
46      set.emplace(2, 2);
47      set.emplace(2, 1);
48      set.emplace(1, 1);
49
50      auto it = set.cbegin();
51      REQUIRE(*it++ == ComparableData(1, 1));
52      REQUIRE(*it++ == ComparableData(1, 2));
53      REQUIRE(*it++ == ComparableData(2, 1));
54      REQUIRE(*it++ == ComparableData(2, 2));
55    }
56  }
```