1 SUV

In dieser Aufgabe wird eine Klassen-Hierarchie bestehend aus den Klassen `Vehicle` und `Suv` aufgebaut. Hierbei soll das Überschreiben von Methoden und der Aufruf von Basis-Konstruktoren ausgenutzt werden.

1.1 Aufgabe

Implementieren Sie beide Klassen, damit beim Ausführen des Test-Programmes folgender Output ausgegeben wird:

```
Vehicle weighs 1200
its speed is 145
```

main.cpp

```cpp
  std::shared_ptr<Vehicle> vehicle = std::make_shared<Suv>(1200, 145);
  std::cout << "Vehicle weighs " << vehicle->GetMass() << std::endl;
  std::cout << "its speed is " << vehicle->GetSpeed() << std::endl;
```

vehicle.h

```cpp
#pragma once

#include <cstddef>

class Vehicle {
 public:
  Vehicle(size_t mass);

  size_t GetMass() const;
  int GetSpeed() const { return 2; };

 private:
  size_t mass_;
};
```

suv.h

```cpp
#pragma once

#include <cstddef>

#include "vehicle.h"

class Suv : public Vehicle {
```

```
8   public:
9    Suv(size_t mass, double speed);
10
11   double GetSpeed() const;
12
13  private:
14   double speed_;
15 };
```

1.2 Lösung

Beim Implementieren der Methoden müssen die folgenden Punkte beachtet werden:

- Die Methode `Vehicle::GetSpeed` muss `virtual` sein und den korrekten Return-Typ zurückgeben.
- Verwenden Sie `override` um dieses Problem einfacher zu erkennen.
- Verwenden Sie im Konstruktor von `Suv` den Basis-Konstruktor: `Vehicle::Vehicle(size_t mass)` um das private Attribut `mass_` zu initialisieren.

vehicle.h

```
1  #pragma once
2
3  #include <cstddef>
4
5  class Vehicle {
6   public:
7    explicit Vehicle(size_t mass);
8
9    size_t GetMass() const;
10   virtual double GetSpeed() const { return 2; }
11
12  private:
13   size_t mass_;
14 };
```

vehicle.cpp

```
1  #include "vehicle.h"
2
3  Vehicle::Vehicle(size_t mass)
4      : mass_(mass) {}
5
6  size_t Vehicle::GetMass() const {
7    return mass_;
8  }
```

suv.h

```h
#pragma once

#include <cstddef>

#include "vehicle.h"

class Suv : public Vehicle {
 public:
  Suv(size_t mass, double speed);

  double GetSpeed() const override;

 private:
  double speed_;
};
```

suv.cpp

```cpp
#include "suv.h"

Suv::Suv(size_t mass, double speed)
    : Vehicle(mass), speed_(speed) {}

double Suv::GetSpeed() const {
  return speed_;
}
```

2 Virtueller Destruktor

In C++ gibt es kein separates Interface-Konzept wie in Java. Man kann Interfaces aber mit virtuellen Methoden erstellen. In dieser Aufgabe sollen Sie die Frage beantworten, weshalb ein virtueller Destruktor in Basis-Klassen eine wichtige Rolle spielt.

2.1 Aufgabe

Kompilieren Sie das gegebene Programm mit und ohne virtuellen Destruktor der Interface-Klasse `BarInterface`, analysieren Sie den Output des Compilers und des Test-Programms und beantworten Sie folgende Fragen:

a) Was ist problematisch an der Zeile a)?
b) Was ist problematisch an der Zeile b)?

main.cpp

```cpp
  // lambda
  auto descriptor = [](BarInterface* obj) {
    obj->Describe();
  };

  std::cout << "BarTester Testing..." << std::endl;
  BarTester* obj1 = new BarTester("Declared with BarTester");
  descriptor(obj1);
  delete obj1;
  obj1 = nullptr;

  std::cout << "BarInterface Testing..." << std::endl;
  BarInterface* obj2 = new BarTester("Declared with BarInterface");
  descriptor(obj2);
  delete obj2;
  obj2 = nullptr;                                    // a)

  std::cout << std::endl << "BarTester not defined..." << std::endl;
  descriptor(new BarTester("Not defined"));          // b)
```

bar_interface.h

```cpp
#pragma once

class BarInterface {
 public:
  virtual ~BarInterface() = default;
  virtual void Describe() = 0;
};
```

bar_tester.h

```cpp
#pragma once

#include <string>

#include "bar_interface.h"

class BarTester : public BarInterface {
 public:
  explicit BarTester(const std::string& name);
  ~BarTester();

  void Describe();

 private:
  std::string private_name_;
```

```
16 };
```

bar_tester.cpp

```
1  #include "bar_tester.h"
2
3  #include <iostream>
4
5  BarTester::BarTester(const std::string& name) : private_name_(name) {
6    std::cout << "BarTester constructor" << std::endl;
7  }
8
9  BarTester::~BarTester() {
10   std::cout << "BarTester destructor" << std::endl;
11 }
12
13 void BarTester::Describe() {
14   std::cout << "I'm BarTester [" << private_name_ << "]" << std::endl;
15 }
```

2.2 Lösung

Der virtuelle Destruktor (in Interfaces typischerweise mittels = `default;` implementiert), stellt sicher, dass der Destruktor der abgeleiteten Klasse aufgerufen wird, auch wenn die Instanz über einen Pointer mit Interface-Typ gelöscht wird.

Folgend die Antworten auf die spezifischen Fragen der Aufgabenstellung.

a) Ohne virtuellen Destruktor wird der Destruktor von `BarTester` nicht aufgerufen, da hier *static Binding* stattfindet.
b) Anonymes Objekt auf dem Heap, welches nicht freigegeben wird.

3 Interfaces und Smart-Pointer

Vererbung und im Speziellen auch Interfaces funktionieren nur mit dynamischem Binding wie gewünscht. Das heisst, es müssen immer Pointer- oder Referenz-Typen für die Funktionsaufrufe verwendet werden. Die Smart-Pointer-Klassen ab C++11 sind ebenfalls darauf ausgelegt und unterstützen dementsprechend implizite Up-Casts und dynamisches Binding.

3.1 Aufgabe

a) Schreiben Sie ein Interface `SocketInterface`, welches folgende Methoden definiert:

```
bool Send(const std::string& message)
bool Receive(std::string* message)
```

b) Erstellen Sie zwei Implementationen dieses Interface: `DummySocket` und `LoopbackSocket`. Der Dummy soll immer senden und auch empfangen können. Er empfängt immer denselben Payload. Der Loopback soll das gesendete in einer `std::deque` speichern und über `Receive` zurückgeben, solange die Queue nicht leer ist.

c) Schreiben Sie einen Unit-Test, welcher einen `std::unique_ptr` vom Typ `SocketInterface` erstellt und diesen mit einer `DummySocket` Instanz füllt. Testen Sie nun die Funktionalität. Danach weisen Sie dem Pointer eine neue Instanz vom Typ `LoopbackSocket` zu und testen dessen Funktionalität.

3.2 Lösung

a) Interface

socket_interface.h

```
#pragma once

#include <string>

class SocketInterface {
 public:
  virtual ~SocketInterface() = default;

  virtual bool Send(const std::string& message) = 0;
  virtual bool Receive(std::string* message) = 0;
};
```

b) Implementationen

dummy_socket.h

```
#pragma once

#include "socket_interface.h"

class DummySocket : public SocketInterface {
 public:
  bool Send(const std::string& message) override;
  bool Receive(std::string* message) override;
};
```

dummy_socket.cpp

```cpp
#include "dummy_socket.h"

bool DummySocket::Send(const std::string& message) {
  return true;
}

bool DummySocket::Receive(std::string* message) {
  *message = "dummy";
  return true;
}
```

loopback_socket.h

```cpp
#pragma once

#include <deque>

#include "socket_interface.h"

class LoopbackSocket : public SocketInterface {
 public:
  bool Send(const std::string& message) override;
  bool Receive(std::string* message) override;

 private:
  std::deque<std::string> fifo_;
};
```

loopback_socket.cpp

```cpp
#include "loopback_socket.h"

bool LoopbackSocket::Send(const std::string& message) {
  fifo_.push_back(message);
  return true;
}

bool LoopbackSocket::Receive(std::string* message) {
  if (fifo_.empty()) {
    return false;
  }
  *message = fifo_.front();
  fifo_.pop_front();
  return true;
}
```

c) Tests

socket_test.cpp

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

#include "dummy_socket.h"
#include "loopback_socket.h"

TEST_CASE("Socket Test", "[SocketInterface, DummySocket, LoopbackSocket]") {
  std::unique_ptr<SocketInterface> socket;
  std::string payload;

  SECTION("DummySocket") {
    socket = std::make_unique<DummySocket>();
    REQUIRE(socket->Send("test"));
    REQUIRE(socket->Receive(&payload));
    REQUIRE(payload == "dummy");
    REQUIRE(socket->Receive(&payload));
    REQUIRE(payload == "dummy");
  }

  SECTION("LoopbackSocket") {
    socket = std::make_unique<LoopbackSocket>();
    REQUIRE(socket->Send("test"));
    REQUIRE(socket->Receive(&payload));
    REQUIRE(payload == "test");
    REQUIRE(socket->Receive(&payload) == false);
  }
}
```

4 Mehrfachvererbung

Generell sollte Mehrfachvererbung vermieden werden, um dadurch erzeugte Probleme zu verhindern. In gewissen Situationen kann sie aber trotzdem hilfreich sein. Z.B. wollen Sie eine `BaseImplementation` erstellen, welche gewisse Basis-Funktionen zur Verfügung stellt. Diese soll auch über ein Interface ansprechbar sein, weshalb Sie ein Interface `BaseInterface` definieren und davon ableiten. Dadurch können Sie nun unterschiedliche Implementationen von `BaseInterface` definieren.

Im weiteren Verlauf, wollen Sie eine erweitere Variante dieses Interface definieren, welche aber auch alle Basis-Funktionen anbietet. Sie leiten also `ExtendedInterface` von `BaseInterface` ab. Natürlich wollen Sie Implementationen zu diesem Interface erstellen, welche sinnvollerweise die Basis-Funktionalität nicht nochmals implementieren, sondern von der Basis erben. Dementsprechend implementieren Sie `ExtendedImplementation` welche von `BaseImplementation` ableitet. Nun haben Sie folgende Klassen-Hierarchie:

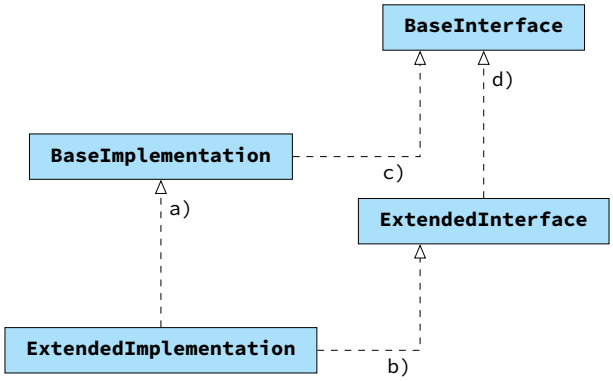

Abbildung 1: Klassen-Hierarchie

4.1 Aufgabe

a) Erstellen Sie alle beschriebenen Klassen mit den entsprechenden Vererbungs-Beziehungen. Als Interface Methode verwenden Sie z.B. `int BaseMethode()` bzw. `int ExtendedMethode()`.

b) Verwenden Sie die Klasse `ExtendedImplementation` über das Interface `ExtendedInterface`. Welche Vererbungs-Beziehungen müssen Sie nun durch *Virtual Inheritance* ersetzten, damit das Program kompiliert?

4.2 Lösung

Sie sollten jede Beziehung zu den Interfaces als `virtual` markieren. Also Beziehung b), c) und d). Beziehung b) ist momentan nicht zwingend. Sollte ihre Klassen-Hierarchie weiter wachsen, wird auch hier *virtual Inheritance* notwendig.

main.cpp

```
1   std::shared_ptr<ExtendedInterface> impl = std::make_shared<
        ExtendedImplementation>();
2   std::cout << "Base: " << impl->BaseMethode() << std::endl;
3   std::cout << "Extended: " << impl->ExtendedMethode() << std::endl;
```

base_interface.h

```
1  #pragma once
2
3  class BaseInterface {
4   public:
5    virtual ~BaseInterface() = default;
6
```

```
7     virtual int BaseMethode() = 0;
8   };
```

base_implemention.h

```
1   #pragma once
2
3   #include "base_interface.h"
4
5   class BaseImplementation : public virtual BaseInterface {
6    public:
7     int BaseMethode() override {
8       return 1;
9     }
10  };
```

extended_interface.h

```
1   #pragma once
2
3   #include "base_interface.h"
4
5   class ExtendedInterface : public virtual BaseInterface {
6    public:
7     virtual ~ExtendedInterface() = default;
8
9     virtual int ExtendedMethode() = 0;
10  };
```

extended_implementation.h

```
1   #pragma once
2
3   #include "extended_interface.h"
4   #include "base_implementation.h"
5
6   class ExtendedImplementation
7       : public virtual ExtendedInterface, public BaseImplementation {
8    public:
9     int ExtendedMethode() override {
10      return 2 + BaseMethode();
11    }
12  };
```