1 Zirkuläre Abhängigkeiten

Smart-Pointer erlauben einfaches und effizientes Memory-Management auf dem Heap. Ein Nachteil der strengen Besitz-Anforderungen von `std::shared_ptr` ist, dass zirkuläre Abhängigkeiten entstehen können. Diese können dazu führen, dass ihre Instanzen nie korrekt zerstört werden. Sollten Sie wichtige Logik in den entsprechenden Destruktoren haben, kann dies zu Problemen führen.

Folgend sind die beiden Klassen `User` und `Bicycle` gegeben.

user.h

```cpp
#pragma once

#include <string>
#include <memory>
#include <unordered_set>

#include "bicycle.h"

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

  std::string GetName() const;
  void UseBicycle(std::shared_ptr<Bicycle> bicycle);
  void PrintUsage() const;

 private:
  const std::string name_;
  std::unordered_set<std::shared_ptr<Bicycle>> used_bicycles_;
};
```

user.cpp

```cpp
#include "user.h"

#include <iostream>

User::User(const std::string& name) : name_(name) {
}

User::~User() {
  std::cout << "dtor: " << GetName() << std::endl;
  PrintUsage();
}

std::string User::GetName() const {
  return name_;
}

void User::UseBicycle(std::shared_ptr<Bicycle> bicycle) {
  used_bicycles_.emplace(bicycle);
}

void User::PrintUsage() const {
  std::cout << "Bicycle usage of: " << name_ << std::endl;
  for (const auto& bicycle : used_bicycles_) {
    std::cout << "  " << bicycle->GetName() << std::endl;
  }
}
```

bicycle.h

```cpp
#pragma once

#include <string>
#include <memory>

class User;

class Bicycle {
 public:
  explicit Bicycle(std::shared_ptr<User> owner);
  ~Bicycle();

  std::string GetName() const;

 private:
  std::string name_;
  std::shared_ptr<User> owner_;
};
```

bicycle.cpp

```cpp
#include "bicycle.h"

#include <iostream>

#include "user.h"

Bicycle::Bicycle(std::shared_ptr<User> owner) : owner_(owner) {
  name_ = std::string("Bicycle of ") + owner_->GetName();
}

Bicycle::~Bicycle() {
  std::cout << "dtor: " << GetName() << std::endl;
}

std::string Bicycle::GetName() const {
  return name_;
}
```

Die vorgegebene Implementation verwendet in beiden Klassen `std::shared_ptr`. Dementsprechend werden diverse Instanzen nie korrekt zerstört, was zu folgendem Output führt:

```
dtor: manuel
Bicycle usage of: manuel
  Bicycle of fabien
  Bicycle of stefan
```

Was allerdings gewünscht wäre, ist der folgende Output:

```
dtor: manuel
Bicycle Usage of: manuel
  Bicycle of fabien
  Bicycle of stefan
dtor: fabien
Bicycle Usage of: fabien
  Bicycle of fabien
dtor: stefan
Bicycle Usage of: stefan
  Bicycle of fabien
  Bicycle of stefan
dtor: Bicycle of fabien
dtor: Bicycle of stefan
```

1.1 Aufgabe

a) Schreiben Sie ein Test-Programm, welches die vorgegebenen Klassen verwendet um den Output 1 zu erzeugen.
b) Korrigieren Sie die Klasse `Bicycle` entsprechend, damit der Output 2 erzeugt wird.

1.2 Lösung

a) Test-Programm

main.cpp

```cpp
  auto stefan = std::make_shared<User>("stefan");
  auto fabien = std::make_shared<User>("fabien");
  auto manuel = std::make_shared<User>("manuel");

  auto stefans_bicycle = std::make_shared<Bicycle>(stefan);
  auto fabiens_bicycle = std::make_shared<Bicycle>(fabien);

  stefan->UseBicycle(stefans_bicycle);
  stefan->UseBicycle(fabiens_bicycle);

  fabien->UseBicycle(fabiens_bicycle);

  manuel->UseBicycle(stefans_bicycle);
  manuel->UseBicycle(fabiens_bicycle);
```

b) Anpassungen Klasse `Bicycle`

Ändern Sie den `std::shared_ptr` in `Bicycle` zu einem `std::weak_ptr`. Zudem passen Sie den Konstruktor fogendermassen an:

```cpp
Bicycle::Bicycle(std::weak_ptr<User> owner) : owner_(owner) {
  name_ = "unknown Bicycle"
  if (auto owner_access = owner_.lock()) {
    name_ = std::string("Bicycle of ") + owner_access->GetName();
  }
}
```

2 Eigener SmartPointer

In dieser Aufgabe sollen Sie eine eigene Implementation eines Reference-Counting Smart-Pointer schreiben. Im Gegensatz zu der Version in der Standard-Library muss ihre Variante nicht generisch und deshalb keine Template-Klasse sein.

Die zu verwaltende Klasse `Item` ist folgend gegeben:

item.h

```cpp
#pragma once

class Item {
 public:
  explicit Item(int number);
  ~Item();

  int GetNumber() const;
  void SetNumber(int number);

 private:
  int number_;
};
```

item.cpp

```
#include "item.h"

#include <iostream>

Item::Item(int number) : number_(number) {
}

Item::~Item() {
  std::cout << "Item: " << number_ << " is destroyed" << std::endl;
}

int Item::GetNumber() const {
  return number_;
}

void Item::SetNumber(int number) {
  number_ = number;
}
```

2.1 Aufgabe

a) Schreiben Sie eine Klasse `ControlBlock` welche die `Item`-Instanz und einen Shared-Count verwaltet.
b) Implementieren Sie die Klasse `SmartPointer` mit folgendem Interface und testen Sie sie in einer kleinen Test-Applikation oder mit Unit-Tests.

smart_pointer.h

```
#pragma once

#include "control_block.h"
#include "item.h"

class SmartPointer {
 public:
  explicit SmartPointer(Item* p = nullptr);
  SmartPointer(const SmartPointer& sp);

  ~SmartPointer();

  SmartPointer& operator=(const SmartPointer& sp);
  SmartPointer& operator=(Item* p);
```

```cpp

  const Item& operator*() const;
  Item* operator->() const;

  bool IsUnique() const;

 private:
  void InitControlBlock(Item* p = nullptr);
  void ReleaseControlBlock();

  ControlBlock* control_block_;
};
```

2.2 Lösung

control_block.h

```cpp
#pragma once

#include <cstddef>

#include "item.h"

class ControlBlock {
 public:
  explicit ControlBlock(Item* instance);

  void Increment();
  void Decrement();

  Item* Get();
  size_t GetSharedCount() const;

 private:
  Item* const instance_;
  size_t shared_count_;
};
```

control_block.cpp

```cpp
#include "control_block.h"

ControlBlock::ControlBlock(Item* instance) : instance_(instance), shared_count_
    (1) {
}

void ControlBlock::Increment() {
  shared_count_++;
}

void ControlBlock::Decrement() {
  if (shared_count_) {
    shared_count_--;

    if (shared_count_ == 0) {
      delete instance_;
    }
  }
}

Item* ControlBlock::Get() {
  return instance_;
}

size_t ControlBlock::GetSharedCount() const {
  return shared_count_;
}
```

smart_pointer.cpp

```cpp
#include "smart_pointer.h"

SmartPointer::SmartPointer(Item* p) {
  InitControlBlock(p);
}

SmartPointer::SmartPointer(const SmartPointer& sp) {
  control_block_ = sp.control_block_;
  if (control_block_) {
    control_block_->Increment();
  }
}

SmartPointer::~SmartPointer() {
  ReleaseControlBlock();
}

SmartPointer& SmartPointer::operator=(const SmartPointer& sp) {
  if (control_block_ != sp.control_block_) {
    ReleaseControlBlock();
    control_block_ = sp.control_block_;
    if (control_block_) control_block_->Increment();
  }
  return *this;
}

```

```cpp
SmartPointer& SmartPointer::operator=(Item* p) {
  ReleaseControlBlock();
  InitControlBlock(p);
  return *this;
}

const Item& SmartPointer::operator*() const {
  return *control_block_->Get();
}

Item* SmartPointer::operator->() const {
  return control_block_->Get();
}

bool SmartPointer::IsUnique() const {
  return control_block_ && control_block_->GetSharedCount() == 1;
}

void SmartPointer::InitControlBlock(Item* p) {
  if (p == nullptr) {
    control_block_ = nullptr;
  } else {
    control_block_ = new ControlBlock(p);
  }
}

void SmartPointer::ReleaseControlBlock() {
  if (control_block_) {
    control_block_->Decrement();
    if (control_block_->GetSharedCount() == 0) {
      delete control_block_;
    }
    control_block_ = nullptr;   // this SmartPointer loses access to the shared
        item
  }
}
```

main.cpp

```cpp
  SmartPointer s;
  assert(!s.IsUnique());
  {
    SmartPointer sp(new Item(11));
    assert(sp.IsUnique());

    s = sp;
    assert(!s.IsUnique());
    assert(!sp.IsUnique());
  }
```

```
    assert(s.IsUnique());
    std::cout << (*s).GetNumber() << std::endl;

    SmartPointer t(nullptr);
    assert(!t.IsUnique());
    t = nullptr;
    assert(!t.IsUnique());
    t = new Item(22);
    std::cout << t->GetNumber() << std::endl;
    assert(t.IsUnique());
    t->SetNumber(33);
    std::cout << t->GetNumber() << std::endl;
    t = nullptr;
    assert(!t.IsUnique());

    SmartPointer u(s);
    assert(!u.IsUnique());
    assert(!s.IsUnique());
```