

## 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

```

1  #pragma once
2
3  #include <string>
4  #include <memory>
5  #include <unordered_set>
6
7  #include "bicycle.h"
8
9  class User {
10 public:
11     explicit User(const std::string& name);
12     ~User();
13
14     std::string GetName() const;
15     void UseBicycle(std::shared_ptr<Bicycle> bicycle);
16     void PrintUsage() const;
17
18 private:
19     const std::string name_;
20     std::unordered_set<std::shared_ptr<Bicycle>> used_bicycles_;
21 };

```

user.cpp

```

1  #include "user.h"
2
3  #include <iostream>
4
5  User::User(const std::string& name) : name_(name) {
6  }
7
8  User::~~User() {
9     std::cout << "dtor: " << GetName() << std::endl;
10    PrintUsage();
11 }
12
13 std::string User::GetName() const {

```

```

14     return name_;
15 }
16
17 void User::UseBicycle(std::shared_ptr<Bicycle> bicycle) {
18     used_bicycles_.emplace(bicycle);
19 }
20
21 void User::PrintUsage() const {
22     std::cout << "Bicycle usage of: " << name_ << std::endl;
23     for (const auto& bicycle : used_bicycles_) {
24         std::cout << "    " << bicycle->GetName() << std::endl;
25     }
26 }

```

## bicycle.h

```

1  #pragma once
2
3  #include <string>
4  #include <memory>
5
6  class User;
7
8  class Bicycle {
9  public:
10     explicit Bicycle(std::shared_ptr<User> owner);
11     ~Bicycle();
12
13     std::string GetName() const;
14
15 private:
16     std::string name_;
17     std::shared_ptr<User> owner_;
18 };

```

## bicycle.cpp

```

1  #include "bicycle.h"
2
3  #include <iostream>
4
5  #include "user.h"
6
7  Bicycle::Bicycle(std::shared_ptr<User> owner) : owner_(owner) {
8     name_ = std::string("Bicycle of ") + owner_->GetName();
9 }
10
11 Bicycle::~Bicycle() {
12     std::cout << "dtor: " << GetName() << std::endl;

```

```
13 }
14
15 std::string Bicycle::GetName() const {
16     return name_;
17 }
```

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

```
1 dtor: manuel
2 Bicycle usage of: manuel
3   Bicycle of fabien
4   Bicycle of stefan
```

Was allerdings gewünscht wäre, ist der folgende Output:

```
1 dtor: manuel
2 Bicycle Usage of: manuel
3   Bicycle of fabien
4   Bicycle of stefan
5 dtor: fabien
6 Bicycle Usage of: fabien
7   Bicycle of fabien
8 dtor: stefan
9 Bicycle Usage of: stefan
10  Bicycle of fabien
11  Bicycle of stefan
12 dtor: Bicycle of fabien
13 dtor: Bicycle of stefan
```

## 1.1 Aufgabe

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

## 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

```

1  #pragma once
2
3  class Item {
4  public:
5      explicit Item(int number);
6      ~Item();
7
8      int GetNumber() const;
9      void SetNumber(int number);
10
11  private:
12      int number_;
13  };

```

item.cpp

```

1  #include "item.h"
2
3  #include <iostream>
4
5  Item::Item(int number) : number_(number) {
6  }
7
8  Item::~Item() {
9      std::cout << "Item: " << number_ << " is destroyed" << std::endl;
10 }
11
12 int Item::GetNumber() const {
13     return number_;
14 }
15
16 void Item::SetNumber(int number) {
17     number_ = number;
18 }

```

## 2.1 Aufgabe

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

smart\_pointer.h

```
1 #pragma once
2
3 #include "control_block.h"
4 #include "item.h"
5
6 class SmartPointer {
7 public:
8     explicit SmartPointer(Item* p = nullptr);
9     SmartPointer(const SmartPointer& sp);
10
11     ~SmartPointer();
12
13     SmartPointer& operator=(const SmartPointer& sp);
14     SmartPointer& operator=(Item* p);
15
16     const Item& operator*() const;
17     Item* operator->() const;
18
19     bool IsUnique() const;
20
21 private:
22     void InitControlBlock(Item* p = nullptr);
23     void ReleaseControlBlock();
24
25     ControlBlock* control_block_;
26 };
```