Motivation

```
class Student {
   string n, v;
   int matNr, alter;
   /* …  */
};
class MatheKlausur{
   vector<Student*>  teilnehmer;
   /*  */
};
class InfoIII_Klausur{
   vector<Student*>  teilnehmer;
   /*  */
};
```

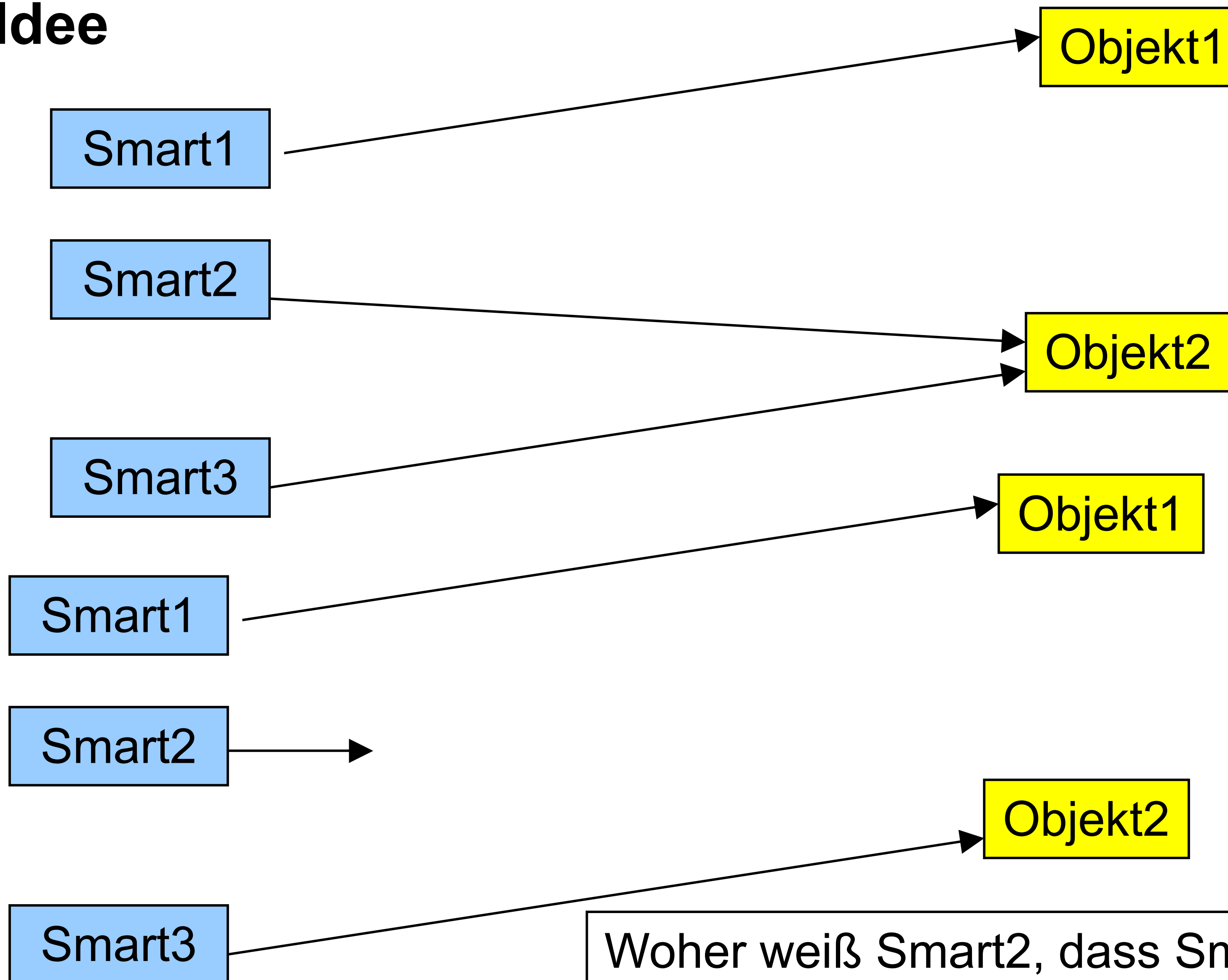
Hier helfen unique_ptr nun überhaupt nicht. Ein und derselbe Student ist in mehreren Instanzen drin, d.h. wenn eine Instanz verschwindet, darf nicht der Student gelöscht werden. Bloß weil es keine MatheKlausur gibt, fällt nicht auch die InfoIII_Klausur aus.

Aufgabe

Implementieren Sie entsprechend der Klasse unique_ptr eine generische Klasse SmartRefPtr, bei der es möglich ist, dass mehrere Zeiger dieses Typs auf das gleiche Objekt zeigen können. Sobald kein Zeiger mehr auf das Objekt verweist, wird es zerstört, d.h. in der Klasse SmartRefPtr muss es einen Zähler geben.

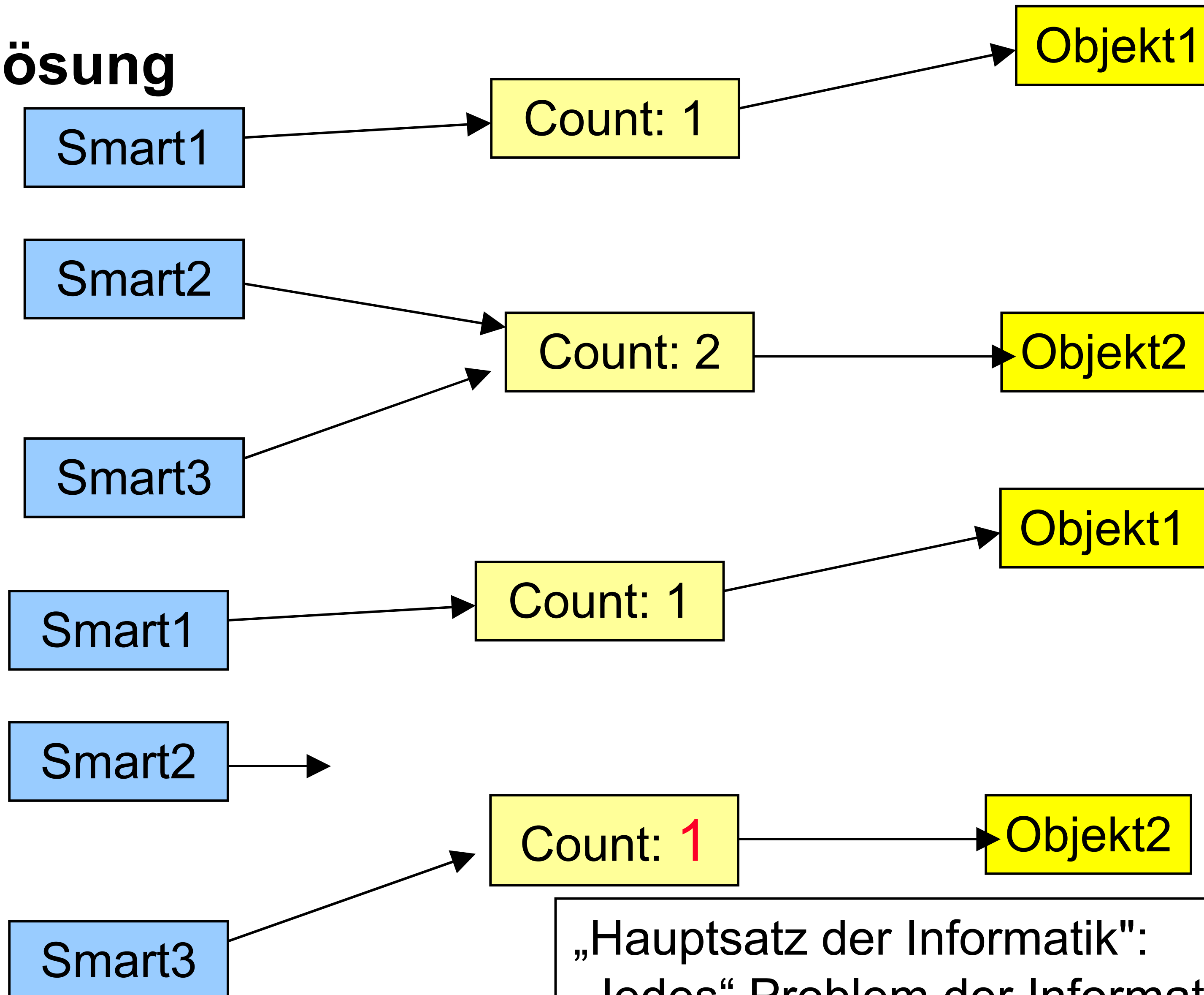
```
int main() {
   SmartRefPtr<int> p1(new int);
   SmartRefPtr<int> p2 = p1;
   if (2 < 3) {
      SmartRefPtr<int> p3;
      p3 = p1;
    // Die drei Zeiger p1, p2, p3 zeigen
    // alle auf das gleiche Objekt
     }
 // Nur noch 2 Zeiger p1, p2 zeigen auf
 // das Objekt
 p2 = new int;  // nur noch einer
 p1 = p2;  // das erste mit new erzeugte
               // Objekt ist zu löschen.
...
 }
```

Idee



Woher weiß Smart2, dass Smart3 noch auf Objekt2 zeigt?

Lösung



„Hauptsatz der Informatik":
„Jedes" Problem der Informatik wird durch eine weitere Indirektionsstufe gelöst

Klassen Count / SmartRefPtr

```
// Vorwaertsreferenz
template <class T> class SmartRefPtr;

template <class T>
class Count {
private:
   Count(T* ap) : p(ap), count(1)
   { cout << "+C " << count << " "; }
   ~Count()  { cout << "-C " << count <<
         " "; }
   int count;
   T* p;
   friend SmartRefPtr<T>;
   Count(const Count&) = delete;
   Count& operator=(const Count&)=delete;
};
```

Korrigierte Version, die auch Count-Instanzen löscht

Klassen Count / SmartRefPtr

```
template <class T>
class SmartRefPtr {
public:
   SmartRefPtr() { p_cnt = 0; }
   SmartRefPtr(T* p) {
      p_cnt = new   Count<T>(p);
   }
   SmartRefPtr(const
       SmartRefPtr& p);
   ~SmartRefPtr() {
      if (p_cnt) {Erniedrige();
       }
   }
   SmartRefPtr&
      operator=(SmartRefPtr& p);
   SmartRefPtr& operator=(T* p);
```

```
private:
   Count<T>* p_cnt;
   void Erhoehe() {
     if (p_cnt){++p_cnt->count;}
   }
   void Erniedrige() {
     if (p_cnt &&
        (--p_cnt->count) == 0) {
         delete p_cnt->p;
         delete p_cnt;
      }
   }
};
```

SmartRefPtr: Implementierung

```
template <class T>
SmartRefPtr<T>::SmartRefPtr(cons
       t SmartRefPtr<T>& p) {
   p_cnt = p.p_cnt;
   Erhoehe();
}
template <class T>
SmartRefPtr<T>&SmartRefPtr<T>:
  operator=(SmartRefPtr<T>& p) {
   if (this != &p) {
      if (p_cnt != nullptr) {
         Erniedrige();
      }
      p_cnt = p.p_cnt;
      Erhoehe();
   }
   return *this;
}
```

```
template <class T>
SmartRefPtr<T>& SmartRefPtr<T>::
  operator=(T* p) {
   if (p_cnt != nullptr) {
      Erniedrige();
   }
   if (p) {
      p_cnt = new Count<T>(p);
   }
   else {
      p_cnt = nullptr;
   }
   return *this;
}
```

SmartRefPtr: Nutzung

```
+T +C 1 +T +C 1 -T -C 0 Ende -T -C 0
```

```
class Test {
public:
   Test() { cout << "+T "; }
   Test(const Test&) { cout <<
       "+TC "; }
   ~Test() { cout << "-T "; }
};
void funk1() {
   SmartRefPtr<Test> pt1(new Test);
   if (2 < 3) {
      SmartRefPtr<Test> pt2 = pt1;
      SmartRefPtr<Test>
                        pt3(new Test);
   }
   cout << "Ende ";
}
```

SmartRefPtr: Nutzung (2)

```
void funk2() {
   SmartRefPtr<Test> p1(new Test);
   SmartRefPtr<Test> p2
   if (2 < 3) {
      SmartRefPtr<Test> p3;
      p3 = p1;
      // Die drei Zeiger p1, p2, p3 zeigen
      // alle auf das gleiche Objekt
   }
   // Nur noch 2 Zeiger p1, p2 zeigen auf das Objekt
   p2 = new Test();  // nur noch einer
   p1 = p2;  // das erste mit new erzeugte Objekt ist zu löschen.
   cout << "Ende ";
}
```

```
+T +C 1 +T +C 1 -T -C 0 Ende -T -C 0
```

Clicker: shared_ptr

```
class Y {public:
   Y(int id=9): m_id(id) {
      cout <<"+Y" << id << " ";}
   ~Y()  {cout <<"-Y" << m_id << " ";}
private:
   int m_id;
};
void ShrPtrUeb1() {
   shared_ptr<Y> p1(new Y(2));
   shared_ptr<Y> p2 = p1;
   if (2 < 3) {
      shared_ptr<Y> p3;
      p3 = p1;
    }
   p2 = shared_ptr<Y>(new Y(8));
   p1 = p2;
}
```

Ausgabe?

1. +Y2 +Y8 -Y2 -Y8
2. +Y2 +Y8 –Y8 –Y2
3. +Y2 + Y2 +Y9 –Y9 +Y8 –Y2 -Y2 -Y8
4. +Y2 +Y9 –Y9 +Y8 -Y2 -Y8

Bitte selber ausprobieren. Code von Homepage herunterladen und Hauptfunktion aus Uebung1.cpp aufrufen

Clicker: shared_ptr

```
class Y {public:
   Y(int id=9): m_id(id) {
      cout <<"+Y" << id << " ";}
   ~Y()  {cout <<"-Y" << m_id << " ";}
private:
   int m_id;
};
void ShrPtrUeb1() {
   shared_ptr<Y> p1(new Y(2));
   shared_ptr<Y> p2 = p1;
   if (2 < 3) {
      shared_ptr<Y> p3;
      p3 = p1;
     }
   p2 = shared_ptr<Y>(new Y(8));
   p1 = p2;
}
```

Clicker: shared_ptr

```
class Y {public:
  Y(int id=9): m_id(id) {
    cout <<"+Y" << id << " ";}
  ~Y()  {cout <<"-Y" << m_id << " ";}
private:
  int m_id;
};
void ShrPtrUeb1() {
  shared_ptr<Y> p1(new Y(2));
  shared_ptr<Y> p2 = p1;
  if (2 < 3) {
    shared_ptr<Y> p3;
    p3 = p1;
   }
  p2 = shared_ptr<Y>(new Y(8));
  p1 = p2;
}
```

Ausgabe?

1. +Y2 +Y8 -Y2 -Y8
2. +Y2 +Y8 –Y8 –Y2
3. +Y2 + Y2 +Y9 –Y9 +Y8 –Y2 -Y2 -Y8
4. +Y2 +Y9 –Y9 +Y8 -Y2 -Y8

1. +Y2 +Y8 -Y2 -Y8

p2 wird zuerst abgeräumt,
Es werden zwar 4 shared_ptr erzeugt, aber nur 2 Y.

2. +Y2 +Y8 –Y8 –Y2
3. +Y2 + Y2 +Y9 –Y9 +Y8 –Y2 -Y2 -Y8
4. +Y2 +Y9 –Y9 +Y8 -Y2 -Y8

Eigentlich sollte ein Programm kein new und delete enthalten

```
void makeSharedUeb9(){
  auto p1 = make_shared<Y>(2);
  auto p2 = p1;
  if (2 < 3) {
    auto p3 = make_shared<Y>();
    p3 = p1;
  }
  p2 = make_shared<Y>(8);
  p1 = p2;
}
```

```
void ShrPtrUeb1(){
  shared_ptr<Y> p1(new Y(2));
  shared_ptr<Y> p2 = p1;
  if (2 < 3) {
    shared_ptr<Y> p3;
    p3 = p1;
  }
  p2 = shared_ptr<Y>(new Y(8));
  p1 = p2;
}
```

```
+Y2 +Y9 -Y9 +Y8 -Y2 -Y8
```

```
+Y2 +Y8 -Y2 -Y8
```

Referenz-Counting

Rekursive Strukturen werden hier in Zusammenhang mit der STL-Schablonen-Klasse weak_ptr auch vorgestellt. Sie werden in der Vorlesung nicht weiter vertieft.
Sie sind hier zum Selbststudium enthalten.

Referenz-Counting und Rekursion
weak_ptr

Clicker: shared_ptr und Rekursion (1)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);    }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

```
void friendTest1(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   mike->addFr(anne);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      anne->addFr(bert);
   }
}
```

Ausgabe?

1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3 -R1 –R3 –R2
4. +R1 +R2 +R3 –R2 –R3 –R1

Siehe Codedatei: Uebung2_Rekursion.cpp

Clicker: shared_ptr und Rekursion (1)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);    }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

```
void friendTest1(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   mike->addFr(anne);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      anne->addFr(bert);
   }
}
```

Clicker: shared_ptr und Rekursion (1)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);    }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

```
void friendTest1(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   mike->addFr(anne);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      anne->addFr(bert);
   }
}
```

Ausgabe?

1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3 -R1 –R3 –R2
4. +R1 +R2 +R3 –R2 –R3 –R1

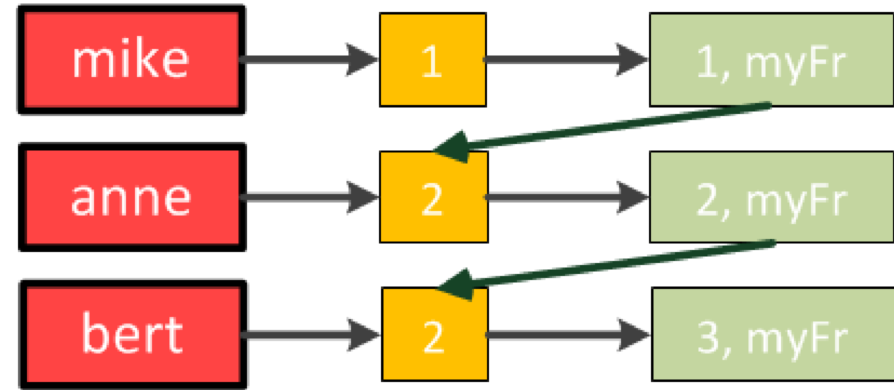
Siehe Codedatei: Uebung2_Rekursion.cpp

1. +R1 +R2 +R3 -R1 -R2 -R3

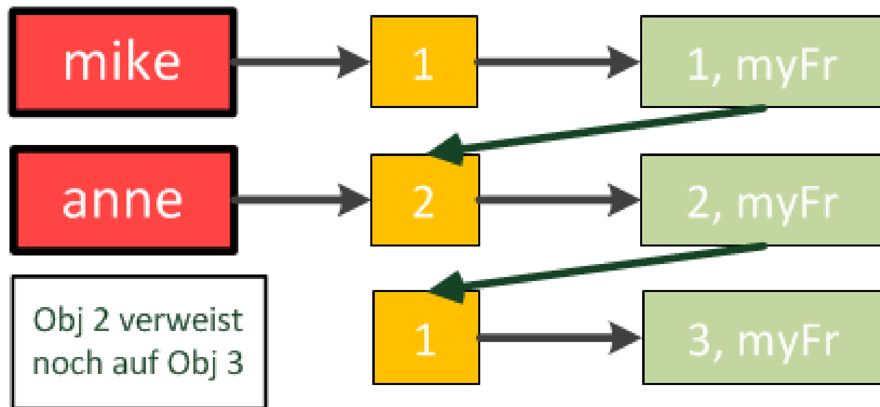
Bert wird freigegeben, noch bei Anne benutzt,
Anne wird freigegeben, noch bei Mike benutzt.
Mike wird freigegeben, damit wird Anne bei Mike richtig freigegeben und damit auch Bert richtig über Anne

```
void friendTest1(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   mike->addFr(anne);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      anne->addFr(bert); // 1
   } // 2
}
```

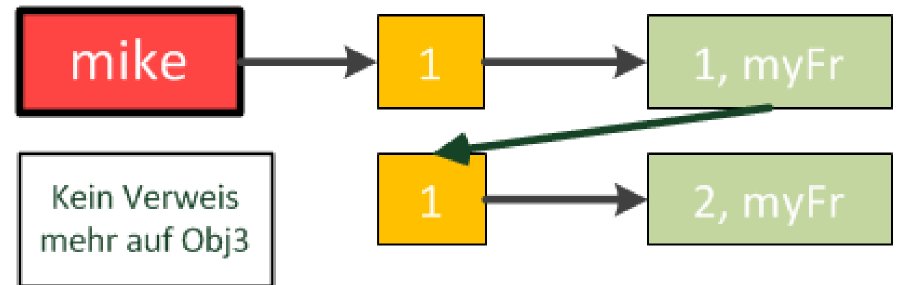
// 1



// bert verschwindet



// anne verschwindet vom Stack



+R1 +R2 +R3   -R1 -R2 -R3

Clicker: shared_ptr und Rekursion (2)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);     }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

```
void friendTest2(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   anne->addFr(mike);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      bert->addFr(anne);
   }
}
```

Ausgabe?

1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3 -R1 –R3 –R2
4. +R1 +R2 +R3 –R2 –R3 –R1

Siehe Codedatei: Uebung3_Rekursion2.cpp

Clicker: shared_ptr und Rekursion (2)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);    }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

```
void friendTest2(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   anne->addFr(mike);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      bert->addFr(anne);
   }
}
```

Ausgabe?

1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3 -R1 –R3 –R2
4. +R1 +R2 +R3 –R2 –R3 –R1

Siehe Codedatei: Uebung3_Rekursion2.cpp

1. +R1 +R2 +R3 –R3 -R2 –R1

Bert wird freigegeben, und Anne auch bei Bert, Bert ist komplett unbenutzt –R3

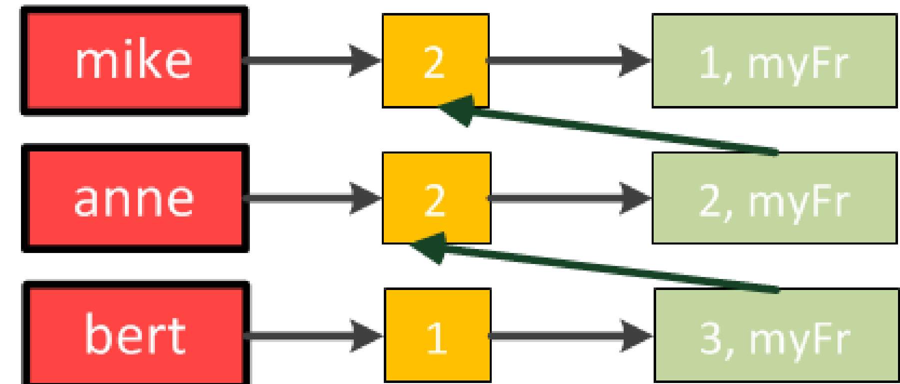
Anne wird freigegeben, und Mike auch bei Anne. Anne ist komplett unbenutzt –R2

Mike wird freigegeben, er ist der letzte –R1

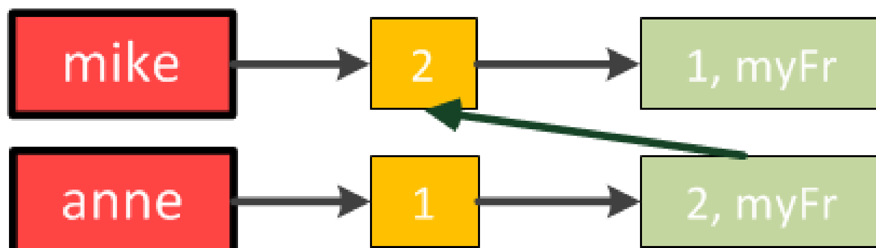
Clicker: shared_ptr und Rekursion (2)

```
void friendTest2(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   anne->addFr(mike);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      bert->addFr(anne); // 1
   } // 2
}
```

// 1



// 2



// anne verschwindet vom Stack

Kein Verweis mehr auf Obj2

+R1 +R2 +R3 -R3 -R2 –R1

Clicker: shared_ptr und Rekursion (3)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);    }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

```
void friendTest3(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   mike->addFr(anne);
   anne->addFr(mike);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      anne->addFr(bert);
   }
}
```

Ausgabe?

1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3
4. +R1 +R2 +R3 –R2 –R3 –R1

Siehe Codedatei: Uebung4_Rekursion3.cpp

Clicker: shared_ptr und Rekursion (3)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);    }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

```
void friendTest3(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   mike->addFr(anne);
   anne->addFr(mike);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      anne->addFr(bert);
   }
}
```

Clicker: shared_ptr und Rekursion (3)

```
class R{
public:
   R(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";  }
   ~R()  {cout<<"-R"<< m_id << " "; }
   void addFr(shared_ptr<R> f){
      myFr.push_back(f);    }
private:
   vector<shared_ptr<R>> myFr;
   int m_id;
};
```

1. +R1 +R2 +R3

```
void friendTest3(){
   shared_ptr<R> mike(new R(1));
   shared_ptr<R> anne(new R(2));
   mike->addFr(anne);
   anne->addFr(mike);
   if (2 < 3){
      shared_ptr<R> bert(new R(3));
      anne->addFr(bert);
   }
}
```

Ausgabe?

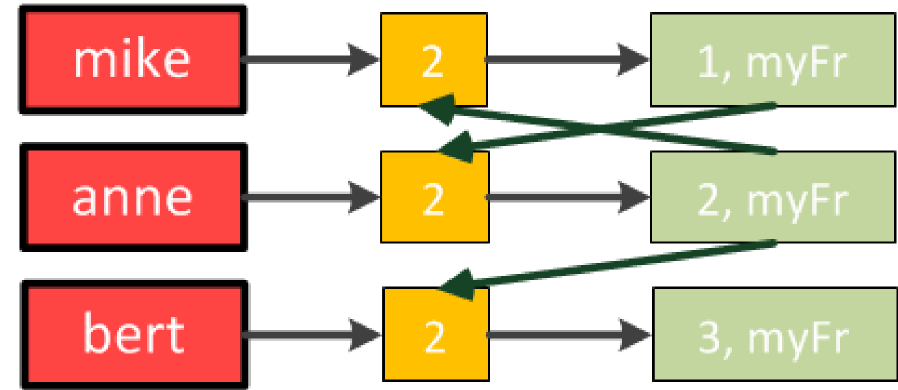
1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3
4. +R1 +R2 +R3 –R2 –R3 –R1

Siehe Codedatei: Uebung4_Rekursion3.cpp

Clicker: shared_ptr und Rekursion (4)

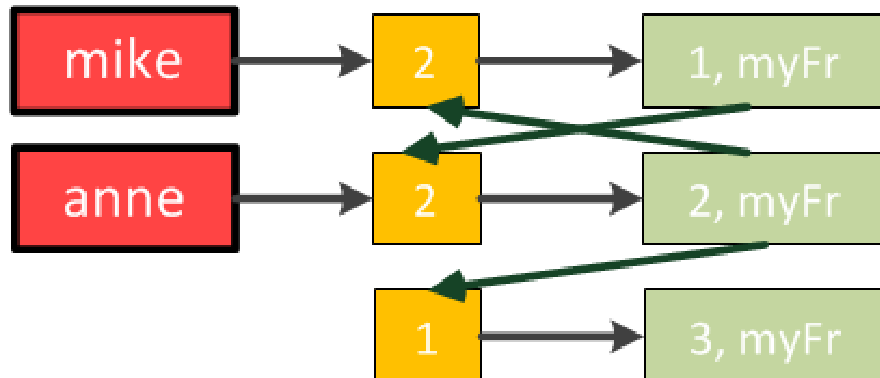
```
void friendTest3(){
  shared_ptr<R> mike(new R(1));
  shared_ptr<R> anne(new R(2));
  mike->addFr(anne);
  anne->addFr(mike);
  if (2 < 3){
    shared_ptr<R> bert(new R(3));
    anne->addFr(bert); //1
  } //2
}
```

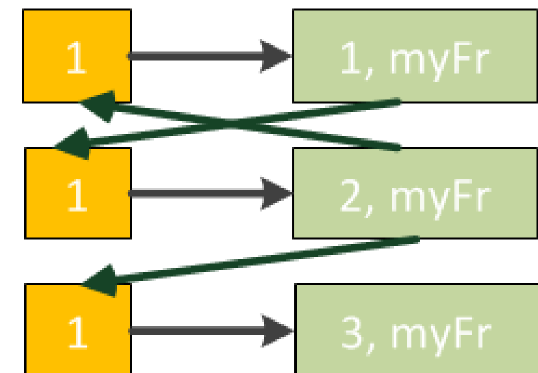
// 1



anne, mike verschwinden vom Stack

// 2





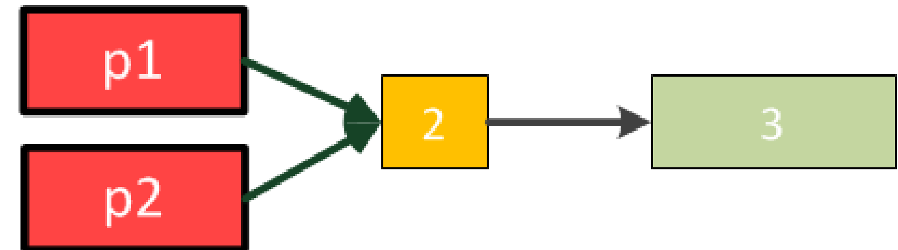
+R1 +R2 +R3

weak_ptr

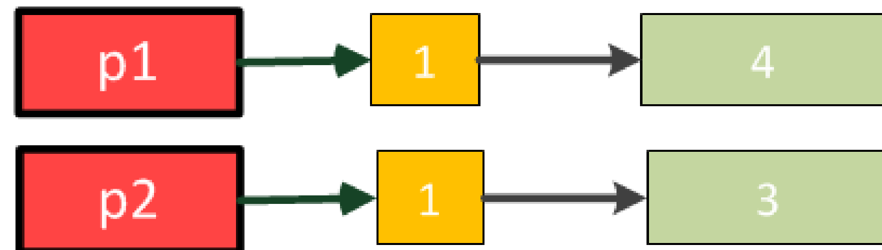
```
#include <memory>

void weakPtr1() {
   shared_ptr<Y> p1(new Y(3));
   shared_ptr<Y> p2 = p1;        // 1
   p1 = shared_ptr<Y>(new Y(4));
   cout << "Ende ";
}// free 3 14
```

Ausgabe:
+Y3 +Y4 Ende -Y3 -Y4

// 1

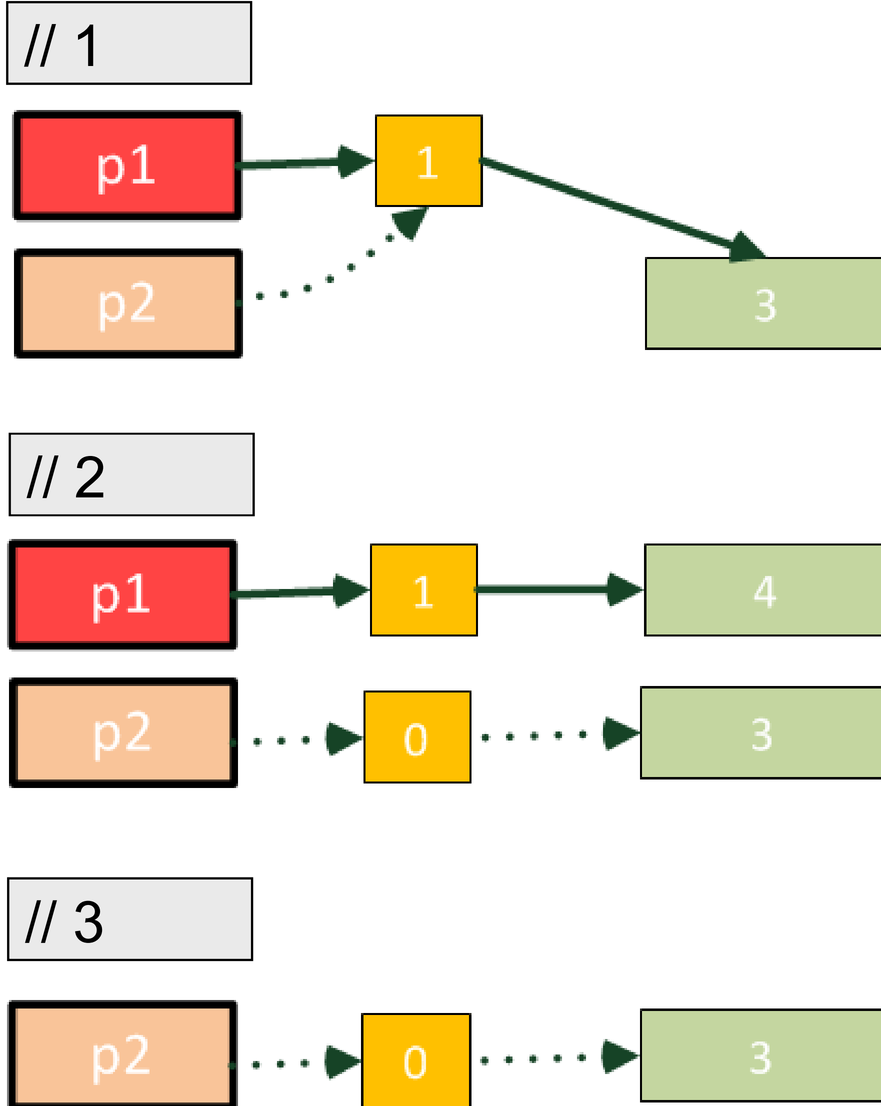


// 2



Siehe Codedatei: Uebung5_weakPtr1.cpp

weak_ptr




```
void weakPtr2() {
   shared_ptr<Y> p1(new Y(3));
   weak_ptr<Y> p2 = p1;                          //1
   p1 = shared_ptr<Y>(new Y(4)); //free 3
   cout << "Ende ";                              //2
}
} // free 4                                      // 3
```

Ausgabe:
+Y3 +Y4 -Y3 Ende -Y4

Siehe Codedatei: Uebung5_weakPtr1.cpp

weak_ptr

```
#include <memory>

void weakPtr1() {
   shared_ptr<Y> p1(new Y(3));
   shared_ptr<Y> p2 = p1;
   p1 = shared_ptr<Y>(new Y(4));
   cout << "Ende ";
}// free 3 14
```

Ausgabe:
+Y3 +Y4 Ende -Y3 -Y4

```
void weakPtr2() {
   shared_ptr<Y> p1(new Y(3));
   weak_ptr<Y> p2 = p1;
   p1 = shared_ptr<Y>(new Y(4)); //free 3
   cout << "Ende ";
} // }// free 14
```

Ausgabe:
+Y3 +Y4 -Y3 Ende -Y4

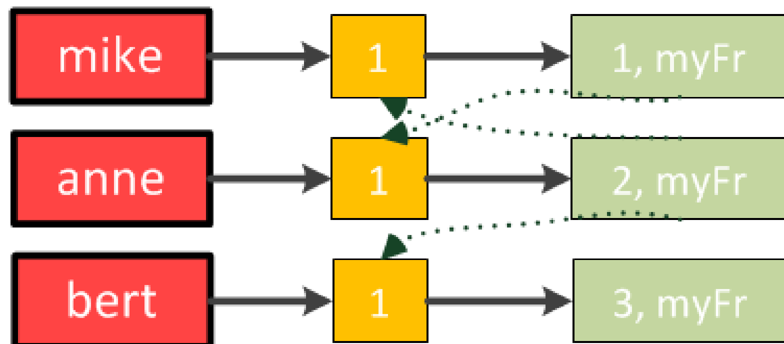
weak_ptr<T> zählen beim Referenzcounting nicht mit. Wenn nur noch weak_ptr auf ein Objekt verweisen, wird das Objekt trotzdem gelöscht. weak_ptr können aber in shared_ptr umgewandelt werden. Verlässt ein weak_ptr<X> seinen Gültigkeitsbereich, wird aber NICHT delete für die Instanz von X aufgerufen.

Clicker: shared_ptr und Rekursion und weak_ptr (1)

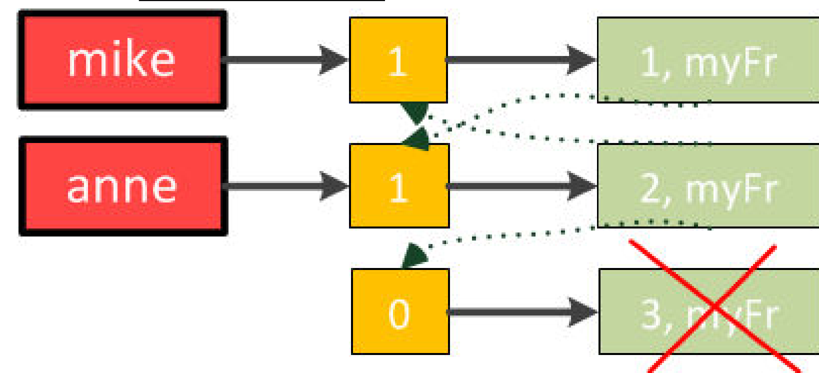
```
class R2{
public:
   R2(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";    }
   ~R2()  {cout << "-R" << m_id << " ";   }
   void addFr(shared_ptr<R2> f){
      myFr.push_back(f);
   }
private:
   vector<weak_ptr<R2>> myFr;
   int m_id;
};
```

```
void friendTestWeak1(){
   shared_ptr<R2> mike(new R2(1));
   shared_ptr<R2> anne(new R2(2));
   mike->addFr(anne);
   anne->addFr(mike);
 if (2 < 3){
      shared_ptr<R2> bert(new R2(3));
      anne->addFr(bert);                  // 1
   }                                       // 2
}
```

// 1



// 2



Siehe Codedatei: Uebung6_weakPtr2.cpp

Clicker: shared_ptr und Rekursion und weak_ptr (2)

```
class R2{
public:
   R2(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";    }
   ~R2()  {cout << "-R" << m_id << " ";   }
   void addFr(shared_ptr<R2> f){
      myFr.push_back(f);
   }
private:
   vector<weak_ptr<R2>> myFr;
   int m_id;
};
```

```
void friendTestWeak1(){
   shared_ptr<R2> mike(new R2(1));
   shared_ptr<R2> anne(new R2(2));
   mike->addFr(anne);
   anne->addFr(mike);
 if (2 < 3){
      shared_ptr<R2> bert(new R2(3));
      anne->addFr(bert);
   }
}
```

Ausgabe?

1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3 -R1 –R3 –R2
4. +R1 +R2 +R3 –R2 –R3 –R1

Clicker: shared_ptr und Rekursion und weak_ptr (2)

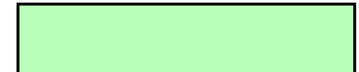
```
class R2{
public:
   R2(int id = 9) : m_id(id) {
      cout << "+R" << id << " ";    }
   ~R2()  {cout << "-R" << m_id << " ";   }
   void addFr(shared_ptr<R2> f){
      myFr.push_back(f);
   }
private:
   vector<weak_ptr<R2>> myFr;
   int m_id;
};
```

2. +R1 +R2 +R3 –R3 -R2 –R1

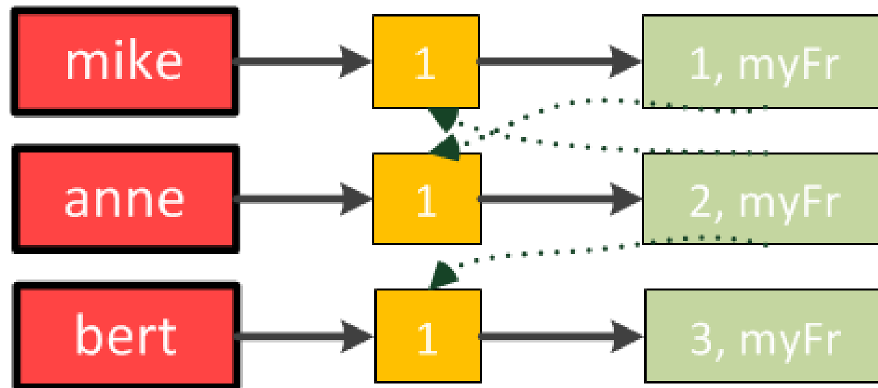
```
void friendTestWeak1(){
   shared_ptr<R2> mike(new R2(1));
   shared_ptr<R2> anne(new R2(2));
   mike->addFr(anne);
   anne->addFr(mike);
 if (2 < 3){
      shared_ptr<R2> bert(new R2(3));
      anne->addFr(bert);
   }
}
```

Ausgabe?

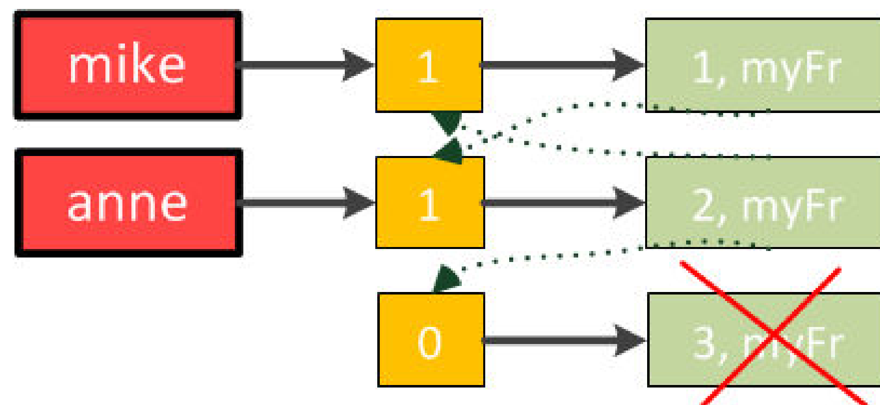
1. +R1 +R2 +R3 -R1 -R2 -R3
2. +R1 +R2 +R3 –R3 -R2 –R1
3. +R1 +R2 +R3 -R1 –R3 –R2
4. +R1 +R2 +R3 –R2 –R3 –R1



Anmerkungen



weak_ptr „wissen“ somit, wie viel shared_ptr noch auf das Objekt zeigen bzw. ob das Objekt überhaupt noch gültig ist (Wert >0) !!!



Clicker: shared_ptr und Rekursion und weak_ptr (3)

```
class R2{
…
void printFriends() const;
private:
   vector<weak_ptr<R2>> myFr;
   int m_id;
};
inline void R2::printFriends() const{
      cout << "friends: ";
      for (vector<weak_ptr<R2>>::const_iterator iter =
                myFr.begin();
          iter != myFr.end(); ++iter) {
         shared_ptr<R2> sp = iter->lock();
         if (sp){
            cout << sp->m_id << " ";
         }
      } // for iter
   }
```

```
void friendTestWeak2(){
   shared_ptr<R2> mike(new R2(1));
   shared_ptr<R2> anne(new R2(2));
   mike->addFr(anne);
   anne->addFr(mike);
 if (2 < 3){
      shared_ptr<R2> bert(new R2(3));
      anne->addFr(bert);
      anne->printFriends();
   }
anne->printFriends();
}
```

Siehe Codedatei: Uebung7_weakPtr3.cpp

Clicker: shared_ptr und Rekursion und weak_ptr (4)

```
class R2{
…
void printFriends() const;
private:
   vector<weak_ptr<R2>> myFr;
   int m_id;
};
inline void R2::printFriends() const{
      cout << "friends: ";
    for (auto iter : myFr) {
         shared_ptr<R2> sp = iter->lock();
         if (sp){
            cout << sp->m_id << " ";
         }
     } // for iter
  }
```

```
void friendTestWeak2(){
   shared_ptr<R2> mike(new R2(1));
   shared_ptr<R2> anne(new R2(2));
   mike->addFr(anne);
   anne->addFr(mike);
 if (2 < 3){
      shared_ptr<R2> bert(new R2(3));
      anne->addFr(bert);
      anne->printFriends();
   }
anne->printFriends();
}
```

```
+R1 +R2 +R3 friends: 1 3 -R3
friends: 1 -R2 -R1
```

Clicker: shared_ptr und Rekursion und weak_ptr (5)

```
class R2{
…
void printFriends(){
      cout << "friends: ";
      for (vector<weak_ptr<R2>>::iterator iter = myFr.begin();
                  iter != myFr.end(); ++iter) {
         shared_ptr<R2> sp = iter->lock();
         if (sp){
            cout << sp->m_id << " ";
         }
      } // for iter
   }
private:
   vector<weak_ptr<R2>> myFr;
   int m_id;
};
```

weak_ptr sind also bei Bedarf in shared_ptr umwandelbar.
Sie verhindern aber auch nicht die Freigabe, wenn sie weak_ptr sind.
Sie behindern sie aber auch nicht.

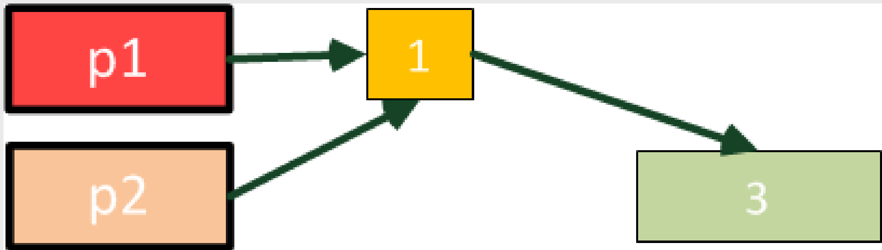
Umwandlung weak_ptr in shared_ptr

Siehe Codedatei: Uebung8_weakPtCount.cpp

```
void weakSharedPtrCount() {
   shared_ptr<Y> p1(new Y(3));                                             //1
   weak_ptr<Y> w2 = p1;
   cout << "cnt " << w2.use_count();                                       // 2
   cout << " w2.i: " << shared_ptr<Y>(w2.lock())->getId()  << " ";         // 3
   p1 = shared_ptr<Y>(new Y(5)); // D 3 ist weg                            //4
   cout << "cnt " << w2.use_count();  // 0                                 // 5
   cout << " Ende ";                                                       // 6
}
```

Ausgabe:
//1 +Y3
/2 cnt 1
//3 w2.i: 3
//4 +Y5 -Y3
//5 cnt 0
//6 Ende
-Y5

// 2



// 5