Scheinaufgabe 1: Fragen zur Datei „telefonbuch.cpp“

(a) **Regel C.62:** Welche Zeilen Code (1-2 Zeilen!!) fehlen im Zuweisungsoperator des Telefonbuchs in „telefonbuch.cpp“? Lesen Sie dafür Regel C.62 nach: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-self (10/2018).

Zeile: 33 Code:
```
... {
if(&telbuch != this){...}
return *this;
}
```

(b) **Const-correctness:** Wenn man in der Methode „get()“ des Telefonbuchs statt „const Eintrag&“ einen „Eintrag&“ zurückgibt:

Compiliert das? [X] Ja, oder [ ] Nein; ist das OK? [ ] Ja, oder [X] Nein

Begründen Sie: Wird const weggelassen, so darf die Adresse des rückgegebenen Pointers verändert werden, d.h. die Einträge im Telefonbuch können somit zerstört/ verändert werden.

Scheinaufgabe 2: Mini-Programmieraufgabe (abzugeben: Code Schnipsel)

(Sie benötigen jeweils nur ein paar wenige Zeilen Code!)

1. Definieren Sie eine Klasse A die nicht kopierbar, aber verschiebbar ist.
2. Definieren Sie eine Klasse B die nicht kopierbar und nicht verschiebbar ist.

Scheinaufgabe 3 (60%): Programmieraufgabe (abzugeben: Code + Begründung)

Ergänzen Sie die Image-Klasse aus der „Übung 2“, so dass man ein Bild kopieren und verschieben kann. Achten Sie dabei auf das korrekte Verhalten:

- Beim Kopieren muss der Bildinhalt (die Pixel) kopiert werden (d.h., jedes Pixel muss kopiert werden).
- Beim Verschieben soll der Bildinhalt (auf den der Pointer auf die Pixel zeigt) „geklaut“ werden.

Zusatzfrage: Warum erfüllt die Image Klasse nicht die „rule of zero“?

Die Klasse trägt nicht nur die Verantwortung für Ownership (Konstruktion und Destruktion), sondern übernimmt auch andere Funktionen wie einen Rahmen zu zeichnen und/oder die Schärfe eines Bildes zu modifizieren

-> Rule of zero verletzt.

Scheinaufgabe 4: Ownership Frage (grob gekoppelt mit der „rule of zero“)

Man sollte man einen shared_ptr / unique_ptr normalerweise lieber mit make_shared / make_unique erstellen und nicht mit dem im folgenden Beispiel gezeigten ebenfalls vorhandenen Konstruktor:

```
#include <iostream>
#include <memory>

struct A {
 A() {std::cout << "A();\n"; }
 ~A() {std::cout << "~A();\n"; }
};

std::unique_ptr<A> f() {
 A* ra=new A();
 // ... more code here! ...
 std::unique_ptr<A> pa = std::unique_ptr<A>{ra}; // take ownersip
 return pa;                                      //       of „ra“
}
```

Was ist hier ein potentielles Problem mit dem hier gezeigten Vorgehen? Was könnte in dem „more code here“-Bereich passieren (bzw. was dürfte man dort nicht machen)? – Überlegen Sie / denken Sie bzgl. Ownership nach.

Der Inhalt von ra könnte im Methodenrumpf freigegeben werden oder zusätzlich anderen Pointern zugewiesen werden, so dass am Ende pa nicht unique sein kann.

Tipp: Die unique_ptr verhindert, dass der Zeiger kopiert wird. Er allein ist verantwortlich für das Freigeben des Speichers. Da es keine Kopien von ihm gibt, ist diese Logik sicher. Denken Sie sich aus, was hier passieren müsste, um mit dieser Logik in Konflikt zu treten...

Scheinaufgabe 5: Warum kompiliert folgender Code nicht?

```
struct A {
  int value=0;
};
struct B {
  int value=0;
  B(int x) : value{x} {}
};
int main() {
  A *a = new A[10];
  B *b = new B[10];
  delete [] a;
  delete [] b;
}
```

Folie: 12

Grund: Der Default-Constructor fehlt und es kann kein Array vom Typ B mit 10 Elementen erzeugt werden.