Übung 1 *Objektverhalten in C++*

```
5  class Zahl
6  {
7  public:
8    Zahl();                 // Konstruktor
9    Zahl( const int& n );   // int-Konstruktor
10   Zahl( const Zahl& n );  // Copy-Konstruktor
11   ~Zahl();                // Destruktor
12   // int-Zuweisung
13   Zahl& operator=( const int& n );
14   // Zuweisung
15   Zahl& operator=( const Zahl& n );
16   // Addition
17   Zahl operator+( const Zahl& n );
18 private:
19   int z;
20 };
21
22 Zahl::Zahl() { z = 0; }
23 Zahl::Zahl( const int& n )  { z = n; }
24 Zahl::Zahl( const Zahl& n ) { z = n.z; }
25 Zahl::~Zahl()               { }
26 Zahl& Zahl::operator=( const int& n )
27 { z=n;   return *this; }
28 Zahl& Zahl::operator=( const Zahl& n )
29 { z=n.z; return *this; }
30 Zahl Zahl::operator+( const Zahl& n )
31 { return Zahl(z+n.z); }
32
33 Zahl f( Zahl a, Zahl& b) { return a+b; }
34
35 int main()
36 {
37   Zahl a(3), b(4);
38   Zahl c(a), d, e;
39   d = 5;
40   d = f(d,c);
41   e = d + a + b;
42   return 0;
43 }
```

Die Klasse `Zahl` ist eine (unvollständige) Implementierung einer Klasse ganzer Zahlen. Geben Sie an, welche Methoden der Klasse `Zahl` bei der Ausführung des obigen Programmes in welcher Reihenfolge aufgerufen werden. Geben Sie dazu für die Zeilen 37–42 an, welche Methoden für welches Objekt aufgerufen werden. Wenn in einer Zeile n der Konstruktor für ein Objekt u, der Zuweisungsoperator für ein Objekt v und der Destruktor für u aufgerufen werden, dann schreiben Sie:

$$n: \qquad u \text{ Konstuktor}, \quad v \text{ Zuweisung}, \quad u \text{ Destruktor}$$

Sie sollen dabei auch *temporäre Objekte* [†] betrachten, die beispielsweise bei der Addition lokal innerhalb der Methode erzeugt werden. Verwenden Sie Großbuchstaben für temporäre Objekte, die in einer Methode erstellt werden.

Beispiel: Der Ausdruck $x = y + z$ erzeugt folgende Aufrufe:

1. Die Additionsmethode von y.
2. Der `int`-Konstruktor in der Additionsmethode erzeugt ein neues Objekt T.
3. Der Copy-Konstruktor erzeugt ein neues Objekt U zur Rückgabe des in der Additionsmethode erzeugten Objekts (denn Rückgabetyp ist `Zahl` und nicht `Zahl&` – eine Referenz zurückzuliefern scheint eine Möglichkeit diese zusätzliche Kopie zu umgehen, wäre aber verhängnisvoll, denn das wäre eine Referenz auf eine lokale temporäre Variable, die bald zerstört wird.)
4. Destruktor des Objektes T.
5. Zuweisungsoperator von x.

[†] In C++ werden temporäre Objekte erzeugt, wenn ein Objekt an eine Methode oder Funktion übergeben wird und wenn eine Methode oder Funktion ein Objekt zurückliefert. In diesem Fall wird nicht das Objekt selbst übergeben, sondern eine Kopie, es sei denn die Übergabe geschieht mit einer Referenz. In einem Ausdruck der Form $a+b+c$ wird zuerst $a+b$ ausgewertet (siehe auch `http://de.cppreference.com/w/cpp/language/operator_precedence`), indem der Additionsoperator für a mit dem Argument b aufgerufen wird. Das Ergebnis wird in einem temporären Objekt gespeichert,dessen Additionsmethode dann mit dem Argument c aufgerufen wird. Objekte werden in umgekehrter Reihenfolge ihrer Erzeugung zerstört.

6. Destruktor von U.

Angenommen, der Ausdruck $x = y + z$ steht in Zeile 41. Dann sieht das Ganze in der Kurzform von oben – der Form, in der Sie die Lösung dieser Aufgabe abgeben sollen – folgendermaßen aus:

41: y Addition T `int`-Konstruktor U Copy-Konstruktor T Destruktor x Zuweisung U Destruktor

Das Programm gibt es unter

`https://conan.iwr.uni-heidelberg.de/data/teaching/info1_ws2019/defmet.cc`

Sie können sich von ihrem Compiler helfen lassen, indem Sie die Methodenaufrufe auf der Konsole ausgeben. Dies geschieht am einfachsten, indem Sie die `print()`-Anweisung in die Methoden der Klasse hineinschreiben. Es gilt jedoch zu beachten, dass Ihr Compiler unter Umständen übereifrig optimiert und dabei unnötiges Aufrufen des Copy-Konstruktors für temporäre Objekte vermeidet, stattdessen aber die Objekte direkt initialisiert. Verwenden Sie die Compileroption `-fno-elide-constructors` um diese Optimierung zu deaktivieren und überprüfen Sie auf jeden Fall, ob Ihre Ergebnisse mit Ihrem Wissen erklärbar sind. **(10 Punkte)**

Übung 2 *Listen-Klasse*

In der Vorlesung haben Sie Funktionen und Datenstrukturen kennengelernt **`int`**-Zahlen in einer einfach verketteten Liste zu verwalten. Die in der Vorlesung vorgestellte Version hat den Nachteil, dass Funktionen und Daten getrennt sind. Basierend auf der Implementierung in den Dateien `intlist.cc` und `intset.cc`, die in den C++-Beispielprogrammen zu finden sind, schreiben Sie deshalb eine Klasse für eine Integerliste. Die Klasse sollte dabei mindestens folgendes Interface erfüllen:

```
class IntList
{
public:
  // Konstruktor, erzeugt eine leere Liste
  IntList();

  // Destruktor, loescht gesamten Listeninhalt
  ~IntList();

  // Gibt Anzahl der Elemente zurueck
  int getCount();

  // Gibt zurueck, ob die Liste leer ist
  bool isEmpty();

  // Gibt die Liste aus
  void print();

  // Fuegt die Zahl 'element' an der (beliebigen) Position 'position' ein
  void insert(int element, int position);

  // Loescht das Element an der Position 'position'
  void remove(int position);

  // Gibt den Wert des Elements an der Position 'position' zurueck
  int getElement(int position);
private:
  // ... (hier folgen private Member der Klasse)
};
```

Dabei bezeichne `position` die Position nach dem ein Element eingefügt oder entfernt werden soll, vgl. mit der Variable `where` aus der Vorlesung.

a) Überlegen Sie sich, welche privaten Daten und Methoden Sie brauchen und welche Funktionen Konstruktor und Destruktor ausführen sollten. Anhand der Methodensignaturen erkennen Sie bereits, dass der Benutzer nur mit `int`s zu tun hat, Dinge wie die Erzeugung von Listenelementen und das Hantieren mit Pointern ist Aufgabe Ihrer Klasse! **[(5 Teilpunkte)]**

b) Implementieren Sie die Interfacemethoden von oben. Es gibt in C++ eine Faustregel namens "Rule of Three" [‡]. Diese Regel besagt, dass die drei Methoden

- Destruktor
- Copy-Konstruktor
- Zuweisungsoperator

immer zusammen auftreten sollten. Implementiert man eine dieser Methoden explizit, sollte man auch die anderen bereitstellen. Die dieser Regel zugrundeliegende Annahme ist folgende: Ist man für eine dieser Methoden mit der impliziten Standardvariante des Compilers nicht zufrieden, trifft dies höchstwahrscheinlich auch auf die anderen beiden Methoden zu. Dies ist meistens dann der Fall, wenn man mit Pointern hantiert.

Implementieren Sie für Ihre Klasse nun also die beiden fehlenden Methoden. Achten Sie bei der Implementierung des

- Copy-Konstruktors darauf, dass Sie eine *tiefe* Kopie[§] Ihrer Liste erstellen. Bloßes Kopieren der Pointer auf die Listenelemente (wie es der Compiler in seiner Default-Variante tun würde) reicht nicht aus, Sie müssen jedes Listenelement neu erstellen!
- Zuweisungsoperators darauf,
  * dass Sie die bereits vorhandenen Daten einer existierenden Liste "aufräumen", bevor Sie ihr neue Daten zuweisen.
  * dass ein Objekt niemals "sich selbst" zugewiesen wird. Den Test, ob das aktuelle Objekt gleich einem als Referenz übergebenen anderen `other` ist, kann man relativ einfach mit `if (this != &other)` überprüfen.

[(5 Teilpunkte)]

Testen Sie Ihre Klasse mit dem folgenden Hauptprogramm:

```
int main()
{
  IntList list;
  list.insert(30);
  list.insert(20);
  list.insert(10);
  list.print();

  list.remove(2);
  list.print();

  list.insert(30,2);
  list.print();

  list.insert(40,3);
  list.print();

  IntList copy(list);
  copy.print();

  copy.remove(0);
  copy.print();
  list.print();
```

[‡] seit C++11 eigentlich "Rule of Five", siehe http://en.wikipedia.org/wiki/Rule_of_three_%28C%2B%2B_programming%29

[§] siehe http://en.wikipedia.org/wiki/Object_copy

```
  copy = list;
  copy.print();

  return 0;
}
```

(10 Punkte)