Aufgabe	A1	A2	A3	A4	$\sum$
Punkte					

Aufgabe 1. Konstruktoren

37:	a int-Konstruktor	b int-Konstruktor			
38:	c Copy-Konstruktor	d Konstruktor	e Konstruktor		
39:	d int-Zuweisung				
40:	T Copy-Konstruktor	T Addition	U int-Konstruktor	V Copy-Konstruktor	U Destruktor
	W Copy-Konstruktor	V Destruktor	d Zuweisung	W Destruktor	T Destruktor
41:	d Addition	T int-Konstruktor	U Copy-Konstruktor	T Destruktor	U Addition
	V int-Konstruktor	W Copy-Konstruktor	V Destruktor	e Zuweisung	W Destruktor
	U Destruktor				
42:	e Destruktor	d Destruktor	c Destruktor	b Destruktor	a Destruktor

Aufgabe 2. (a) Der Implementierung liegt weiterhin eine einfach verkettete Liste zu Grunde. Konkret wird die `struct IntListElem` verwendet.

Es wird ein Zeiger `IntListElem* first` auf das erste Listenelement, eine Variable `int count`, die die Länge der Liste speichert und zwei Hilfsmethoden, `IntListElem* getListElement(int position)`, die einen Zeiger auf das interne Listenelement zurückgibt und `void copy(IntList& other)`, die die Listenelemente von einer anderen Liste kopiert, verwendet.

Der leere Konstruktor setzt `first` und `count` auf 0, genauso wie der Destruktor. Dies führt zur Initialisierung bzw. Leerung der Liste. Der Copy-Konstruktor bzw. Zuweisungsoperator initialisieren bzw. leeren die Liste ebenfalls zunächst und kopieren dann mithilfe der `copy` Methode, die Elemente einer zweiten Liste.

(b)
```
#include<stdio.h>

// a list element
struct IntListElem {
    int value; // int value
    IntListElem* next; // pointer to the next element
};

// A list of integers
// Beware: indexing is done starting from 1 as the excercise sheet requests it
class IntList {
public:
    // empty constructor, creates an empty list
    IntList();

    // copy constructor, intializes by copying from 'other'
    IntList(IntList& other);

    // assignment operator, removes own elements and copies from 'other'
    IntList& operator=(IntList& other);

    // destructor, removes list content
    ~IntList();

    // returns the number of elements in the list
    int getCount();

    // returns if the list is empty
    bool isEmpty();

    // prints the list
    void print();

    // inserts 'element' in the beginning of the list
    void insert(int element);

    // inserts 'element' AFTER 'position'
    void insert(int element, int position);

    // removes the element AFTER 'position'
    void remove(int position);

```

```
43      // returns the element AT 'position'
44      int getElement(int position);
45  private:
46      // pointer to first element
47      IntListElem* first;
48      // element count
49      int count;
50
51      // returns the IntListElem AT 'position'
52      IntListElem* getListElement(int position);
53
54      // Kopiert die Liste 'other'
55      void copy(IntList& other);
56  };
57
58  // creates new empty list
59  IntList::IntList() {
60      first = 0;
61      count = 0;
62  }
63
64  // creates a new list by copying an old one
65  IntList::IntList(IntList& list) {
66      copy(list);
67  }
68
69  // Assignment operator
70  IntList& IntList::operator=(IntList& other) {
71      if (this == &other) { // if assigning to the same object
72          return *this; // do nothing
73      }
74      copy(other); // otherwise copy contents
75      return *this;
76  }
77
78  // destroys the list
79  IntList::~IntList() {
80      first = 0; // remove the connection to the list elements
81      count = 0; // reset list count
82  };
83
84  // private helper that copies from 'other' list
85  void IntList::copy(IntList& other) {
86      // clear and copy elements
87      first = 0;
88      count = 0;
89      // copy element by element from 'other' to this new list
90      for (int i = 1; i <= other.count; i++) {
91          insert(other.getElement(i), i-1);
92      }
93  }
94
95  // private helper that returns the list element struct AT 'position'
96  IntListElem* IntList::getListElement(int position) {
97      IntListElem* current = first;
98      for (int i = 1; i < position; i++) {
99          // if list is not long enough
100         if (current->next == 0) {
101             // return the last element
102             return current;
103         }
104         current = current->next;
105     }
106     return current;
107 }
108
109 // return element AT 'position'
110 int IntList::getElement(int position) {
111     if (position < 1 || position > count) {
112         printf("Error: Index out of Bounds, no element at position %d\n", position)
    ;
113         return -1;
114     }
115     return getListElement(position)->value; // return the value of the element
116 }
117
```

```
118 // insert 'element' AFTER 'position'
119 void IntList::insert(int element, int position) {
120     if (position < 0 || position > count) { // check for invalid positions
121         printf("Error: Index out of Bounds, can't insert.\n");
122         return;
123     }
124     IntListElem* elem = new IntListElem(); // create a new list element
125     elem->value = element; // with the given value
126     if (position == 0) { // insert in the very beginning
127         elem->next = first; // the successor of elem is the old first, now second
    element
128         first = elem; // make the new element the new first
129     } else {
130         IntListElem* where = getListElement(position); // find the element to
    insert after
131         elem->next = where->next; // make the old successor of where now follow
    elem
132         where->next = elem; // make elem the new successor of where
133     }
134     count += 1;
135     return;
136 }
137
138 // insert 'element' in the very beginning of the list
139 void IntList::insert(int element) {
140     insert(element, 0); // insert AFTER position 0
141 }
142
143 // remove next element AFTER 'position'
144 void IntList::remove(int position) {
145     if (position < 0 || position >= count) { // check for invalid positions
146         printf("Error: Index out of bounds, can't remove after %d\n", position);
147         return;
148     }
149     if (position == 0) { // remove first element
150         first = first->next; // new first is the new element
151     } else {
152         IntListElem* where = getListElement(position); // find the element to
    remove after
153         where->next = where->next->next; // remove by skipping the next element of
    where
154     }
155     count -= 1; // reduce element count
156     return;
157 }
158
159 // check if list is empty
160 bool IntList::isEmpty() {
161     return count == 0;
162 }
163
164 // returns the list's size
165 int IntList::getCount() {
166     return count;
167 }
168
169 // print a list of integers
170 void IntList::print() {
171     printf("["); // opening brackets
172     IntListElem* current = first;
173     while (current != 0) { // while not last element
174         printf("%d", current->value); // print value
175         if (current->next != 0) {
176             printf(", "); // if not the last element, append a comma
177         }
178         current = current->next; // go to next
179     }
180     printf("]\n"); // closing brackets
181 }
182
183 int main() {
184     IntList list;
185     list.insert(30);
186     list.insert(20);
187     list.insert(10);
188     list.print();
```

```
189
190     list.remove(2);
191     list.print();
192
193     list.insert(30,2);
194     list.print();
195
196     list.insert(40,3);
197     list.print();
198
199     IntList copy(list);
200     copy.print();
201
202     copy.remove(0);
203     copy.print();
204     list.print();
205
206     copy = list;
207     copy.print();
208
209     return 0;
210 }
211
```

listclass.cpp