Aufgabe 2: Dynamische Arrays in C++11

Ihre Aufgabe ist die Implementierung einer Klasse zur Verwaltung von dynamischen Arrays. Die Elemente in dem dynamischen Array sind alle vom gleichen aber beliebigen Typ. Des verwenden wir eine templatifizierte Klasse.

Das Design der Klasse orientiert sich an dem Design der "String" Klasse bekannt aus der Vorlesung und soll dem "Rule of Five" Designmusster (copy/move Semantik) folgen.

Es folgt eine Beschreibung der zu implementierenden Konstruktoren/Methoden.

Konstruktoren/Zuweisungsmethode (copy/move)

```
DynArr(const DynArr<T>& src) {
 // TODO
}
 DynArr<T>& operator=(const DynArr<T>& src) {
 // TODO
}
DynArr(DynArr<T>&& src) {
 // TODO
}
DynArr<T>& operator=(DynArr<T>&& src) {
 // TODO
}
```

Das zu implementierende Verhalten ist analog zur "String" Klasse.

add Methode

```
void add(T x)
```

Füge Element x hinzu (an den Schluss).

Hinweise.

- Lege neuen Speicher an (bisherige Größe + 1)
- Kopiere bestehende Elemente, wobei x an den Schluss kommt
- Lösche bestehenden Speicher

reverse Methode

```
void reverse()
```

Bilde die Umkehrung. Letztes Element wird erstes Element usw.

Hinweise.

- Codefragmente aus Aufgabe 1 dürfen übernommen werden

append Methode

```
void append(const DynArr<T>& src)
```

Füge eine dynamisches Array src hinzu (an den Schluss).

Hinweise.

- Lege neuen Speicher an (bisherige Größe + Größe von src)
- Kopiere bestehende Elemente, wobei Elemente von src an den Schluss kommen
- Lösche bestehenden Speicher

Mit Hilfe von append kann man add implementieren.

Anhang: Source Files

dynArr.hpp

```
#ifndef __DYNARR__
#define __DYNARR__

#include <iostream>
using namespace std;

template<typename T>
void copy_(T* s, int n, T* t) {
  int i = 0;
  while(i < n) {
    t[i] = s[i];
    i++;
  }
}


template<typename T>
class DynArr {
  int len;
  T* p;

public:

  DynArr() {
    this->len = 0;
    this->p = nullptr;
  }
  DynArr(T x, int size) {
    this->len = size;
    this->p = new T[size];
```

```
    for(int i=0; i<size; i++) {
      this->p[i] = x;
    }
  }
  DynArr(const DynArr<T>& src) {
   // TODO
  }
   DynArr<T>& operator=(const DynArr<T>& src) {
   // TODO
  }
  DynArr(DynArr<T>&& src) {
   // TODO
  }
  DynArr<T>& operator=(DynArr<T>&& src) {
   // TODO
  }


  ~DynArr() { delete[] p; }

  T& operator[](int index) {
    return p[index];
  }


  int size() const {
      return this->len;
  }

  string show() {
    string s;
    for(int i=0; i<this->len; i++) {
      s = s + to_string(p[i]);
    }
    return s;
  }


  void add(T x) {
   // TODO
  }

  void reverse() {
  // TODO
  }

  void append(const DynArr<T>& src) {
  // TODO
  }

};
```

```
#endif
```

testDynArr.cpp

```
#include <iostream>
using namespace std;

#include "dynArr.hpp"

void example1() {

  DynArr<int> d;

  d.add(1);

  cout << "\n1. " << d[0];

  d.add(2);

  cout << "\n2. " << d[0] << d[1];

  cout << "\n3. " << d.show();

  d.reverse();

  cout << "\n4. " << d.show();

  d.append(d);

  cout << "\n5. " << d.show();

  DynArr<int> d2;

  d2 = d;

  cout << "\n6. " << d2.show();

  d.reverse();

  d2.append(d);

  cout << "\n7. " << d2.show();

}


template<typename T>
void someFunc(int i, DynArr<T> d) {
  d.reverse();
```

```
  cout << "\n" << i << ". " << d.show();
}


void example2() {

  DynArr<bool> d = DynArr<bool>();

  d.add(true);

  cout << "\n1. " << d[0];

  d.add(false);

  cout << "\n2. " << d[0] << d[1];

  cout << "\n3. " << d.show();

  someFunc<bool>(3,d);

  cout << "\n4. " << d.show();

  someFunc<bool>(5,DynArr<bool>(true, 2));

  DynArr<bool> d2;

  d2 = move(d);

  d2.add(false);

  cout << "\n6. " << d2.show();



}


int main() {

  cout << "\n\n *** example1 *** \n";
  example1();

  cout << "\n\n *** example2 *** \n";
  example2();


}
```