**Programmiersprachen**
**Sommersemester 2015**

# Aufgabensammlung 5

Die Aufgaben werden am **15. Juni** in der Übung bewertet. Diese Aufgabensammlung beschäftigt sich mit Pointern, Containern und der Standard Template Library (STL). Es gelten die Ausführungshinweise des vorherigen Aufgabenblattes (`const`-Korrektheit, initialization-list, Header/Source, CMakeLists.txt, catch.hpp, ...). Nutzen Sie neben dem Vorlesungsskript ausschließlich aktuelle Fachliteratur oder Online-Referenzen, z.B.

- Stroustrup, B.: Einführung in die Programmierung mit C++ (2010)
- http://en.cppreference.com/
- http://www.cplusplus.com/
- http://en.cppreference.com/w/cpp/container/list

Bei Fragen und Anmerkungen schreiben Sie bitte eine Email an andreas.bernstein@uni-weimar.de .

## Aufgabe 5.1

Implementieren Sie eine doppelt verkettete Liste namens `List` als Template. Verwenden Sie bitte den hier vorgegebenen Header `List.hpp`. Das struct `ListNode` enthält Zeiger auf das vorherige, das nächste Element und hält ein Objekt vom Typ `T`.

```
#ifndef BUW_LIST_HPP
#define BUW_LIST_HPP

#include <cstddef>

// List.hpp

template <typename T>
struct List;

template <typename T>
struct ListNode
{
  ListNode() : m_value(), m_prev(nullptr), m_next(nullptr) {}
```

```
  ListNode(T const& v, ListNode* prev, ListNode* next)
    : m_value(v), m_prev(prev), m_next(next)
  {}
  T m_value;
  ListNode* m_prev; //
  ListNode* m_next;
};

template <typename T>
struct ListIterator
{
  friend class List<T>;
// not implemented yet
private:
  ListNode<T>* m_node;
};

template <typename T>
struct ListConstIterator
{
public:
// not implemented yet
private:
  ListNode<T>* m_node;
};

template <typename T>
class List
{
public:
// not implemented yet
private:
  std::size_t m_size;
  ListNode<T>* m_first;
  ListNode<T>* m_last;
};

#endif // #define BUW_LIST_HPP
```

Die Listenenden werden durch einen Nullzeiger `nullptr` gekennzeichnet.

In dieser Aufgabe implementieren Sie den Standard-Konstruktor, `empty` und `size`, also folgende public Methoden.

```
// Default Constructor
List();
// http://en.cppreference.com/w/cpp/container/list/empty
bool empty() const;
// http://en.cppreference.com/w/cpp/container/list/size
std::size_t size() const;
```

Die Methode `size` gibt die Länge der Liste zurück. Neben dem Header `List.hpp` brauchen Sie nur eine weitere Datei, die Sie `TestList.cpp` nennen können. Diese enthält dann ihre Tests. **[10 Punkte]**

## Aufgabe 5.2

Implementieren Sie nun die Methoden `push_front`, `push_back`, `pop_front`, `pop_back`, `front` und `back`. Testen Sie alle. Ein Test könnte so aussehen.

```
TEST_CASE("add an element with push_front", "[push_front]")
{
  List<int> list;
  list.push_front(42);
  REQUIRE(42 == list.front());
}
```

**[15 Punkte]**

## Aufgabe 5.3

Schreiben Sie die Methode `clear`, welche alle Elemente der Liste löscht. Am besten verwenden Sie dazu eine der `pop_*` Methoden.

```
TEST_CASE("should be empty after clearing", "[clear]")
{
  List<int> list;
  list.push_front(1);
  list.push_front(2);
  list.push_front(3);
  list.push_front(4);
  list.clear();
  REQUIRE(list.empty());
}
```

Implementieren Sie den Destruktor mit `clear()`. **[5 Punkte]**

## Aufgabe 5.4

Die Verwendung von Iteratoren kennen Sie bereits aus den letzten beiden Aufgabenblättern. In dieser Aufgabe sollen Sie selber einen implementieren. Fügen Sie dazu folgendes Template in die Datei `List.hpp` über der Definition der `List` ein. Implementieren Sie alle Methoden die als *// not implemented yet* markiert sind. Um die Lesbarkeit zu verbessern, haben wir einige typedefs verwendet. (http://en.cppreference.com/w/cpp/language/typedef)

```
template <typename T>
struct ListIterator
{
  typedef ListIterator<T> Self;
  typedef ListNode<T> Node;
  typedef T value_type;
  typedef T* pointer;
  typedef T& reference;
  typedef ptrdiff_t difference_type;
  typedef std::forward_iterator_tag iterator_category;

  friend class List<T>;

  ListIterator() {} // not implemented yet
  ListIterator(ListNode<T>* n) {} // not implemented yet
  reference operator*() const {} // not implemented yet
  pointer operator->() const {} // not implemented yet
  Self& operator++() {} // not implemented yet
  Self operator++(int) {} // not implemented yet
  bool operator==(const Self& x) const {} // not implemented yet
  bool operator!=(const Self& x) const {} // not implemented yet
  Self next() const
  {
    if (m_node)
      return ListIterator(m_node->m_next);
    else
      return ListIterator(nullptr);
  }
private:
  // The Node the iterator is pointing to
  ListNode<T>* m_node;
};
```

**[15 Punkte]**

## Aufgabe 5.5

Schreiben Sie nun die Methoden `begin` und `end`, welche Iteratoren auf das erste Knotenelement bzw. einen mit nullptr initialisierten Iterator zurückgeben. Testen Sie danach das Interface wie folgt.

```
TEST_CASE("should be an empty range after default construction",
          "[begin_end]")
{
  List<int> list;
  auto b = list.begin();
  auto e = list.end();
  REQUIRE(b == e);
}

TEST_CASE("provide acces to the first element with begin", "[begin]")
{
  List<int> list;
  list.push_front(42);
  REQUIRE(42 == *list.begin());
}

// TEST_CASE ...
```

**[5 Punkte]**

## Aufgabe 5.6

Implementieren Sie nun folgende Funktionen.

```
template<typename T>
bool operator==(List<T> const& xs, List<T> const& ys)
{
  // not implemented yet
}

template<typename T>
bool operator!=(List<T> const& xs, List<T> const& ys)
{
  // not implemented yet
}
```

**[5 Punkte]**

## Aufgabe 5.7

Implementieren Sie nun den Copy-Konstruktor und den Move constructor. Überlegen und beschreiben Sie, wie der Compiler den Copy-Konstruktor erzeugt und was bei dessen Ausführung geschieht. Implementieren Sie nun selber den Copy-Konstruktor. Dieser soll eine komplette Listenkopie erstellen. Verwenden Sie folgende Tests.

```
TEST_CASE("copy constructor", "[constructor]")
{
  List<int> list;
  list.push_front(1);
  list.push_front(2);
  list.push_front(3);
  list.push_front(4);
  List<int> list2(list);
  REQUIRE(list == list2);
}

TEST_CASE("move constructor", "[constructor]")
{
  List<int> list;
  list.push_front(1);
  list.push_front(2);
  list.push_front(3);
  list.push_front(4);
  List<int> list2(std::move(list));
  REQUIRE(0 == list.size());
  REQUIRE(list.empty());
  REQUIRE(4 == list2.size());
}
```

Protip: For den Copy-Konstruktor können Sie die `push_*` Methoden verwenden.

**[10 Punkte]**

## Aufgabe 5.8

Schreiben Sie die Methode `insert` zum Einfügen eines Objekts vor einer angegeben Position. Als Argumente dieser Methode werden die Position als Iterator,

und das einzufügende Objekt übergeben. Möglicherweise müssen Sie den ListIterator anpassen. **[10 Punkte]**

## Aufgabe 5.9

Implementieren Sie die Methode `reverse`, welche die Reihenfolge der Liste umkehrt. Als zweites implementieren Sie die freie Funktion `reverse`, welche eine Liste als Argument bekommt und ein neue Liste mit umgekehrter Reihenfolge zurückgibt. Testen sie diese Methode ausreichend! **[10 Punkte]**

## Aufgabe 5.10

Implementieren Sie den Zuweisungsoperator (wie in der Vorlesung vorgestellt). **[5 Punkte, optional]**

## Aufgabe 5.11

Verwenden Sie `std::copy` um eine `List<int>` in einen `std::vector<int>` zu kopieren. **[5 Punkte, optional]**

Bei Fragen und Anmerkungen schreiben Sie bitte eine Email an andreas.bernstein@uni-weimar.de .