

Aufgabensammlung 4

Die Aufgaben werden am Mittwoch den **7. 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 (github-Repository, **const**-Korrektheit, initialization-list, Header/Source, CMakeLists.txt, catch.hpp, ...). Commiten Sie nach jedem implementierten Feature. 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://en.cppreference.com/w/cpp/container/list>

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

Aufgabe 4.1

Erstellen Sie unter ihrem eigenen *github*-Account ein neues Repository mit dem Namen *programmiersprachen-aufgabenblatt-4*. Klonen Sie das erstellte Repository. Erstellen Sie darin die gleiche Struktur wie in den letzten Repositories.

Ein wichtiges Werkzeug zur Fehlersuche und Analyse von Programmen ist der Debugger. Diese Aufgabe soll Ihnen das Debuggen am Beispiel eines einfachen Programms zeigen.

Erstellen Sie den Ordner *build* und konfigurieren Sie ihr Projekt im Debug-Modus. Dazu wird bei CMake die Option `CMAKE_BUILD_TYPE` auf `DEBUG` gesetzt. Die Option können Sie über die GUI direkt setzen oder auf der Kommandozeile als Parameter wie folgt angeben:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Wer Visual Studio verwendet, nutzt den Visual-Studio-Debugger.

Fügen Sie folgendes Beispielprogramm hinzu und kompilieren Sie es.

```
#include <iostream>
```

```
int compute()
{
    int val = 1111;
```

```

    std::cout << "computed: " << val << '\n';
    return val;
}

int main(int argc, char *argv[])
{
    int key = compute();
    if (1234 != key) {
        std::cout << "ERROR: wrong key\n";
        return 1;
    }
    std::cout << "Achievement unlocked\n";
    return 0;
}

```

Bauen Sie das Programm und führen Sie es aus. Wir wollen die Variablen des Programm zur Laufzeit ändern, so dass *Achievement unlocked* ausgegeben wird. Dazu rufen Sie den Debugger auf und setzen einen *breakpoint* auf *main*. Mit GDB sieht das wie folgt aus:

```

gdb aufgabe1
break main
run

```

Folgende Befehle stehen Ihnen nun zur Verfügung.

break, b set break point, e.g. **break** main

run, r run program

print, p print variable, e.g. **print** val

set set variable, e.g. **set** val = 1000

step, s step program to next source line

backtrace, bt summary of how your program got where it is, shows frame number and function name

Ihre Aufgabe ist es nun im Debugger *val* auf *1234* zu setzen. Um die ausgeführten Schritte dieses Programmes zu verstehen, verwenden Sie den Befehl **step**. Bereiten Sie die Aufgabe so vor, dass Sie sie zur Übung vorführen können.

[5 Punkte]

Aufgabe 4.2

Implementieren Sie eine doppelt verkettete Liste namens *List* als Template. Verwenden Sie 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 = nullptr;
};

template <typename T>
struct ListConstIterator
{
    friend class List<T>;
public:
    // not implemented yet
private:
};
```

```

    ListNode<T>* m_node = nullptr;
};

template <typename T>
class List
{
public:
    typedef T value_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef ListIterator<T> iterator;
    typedef ListConstIterator<T> const_iterator;

    friend class ListIterator<T>;
    friend class ListConstIterator<T>;
    // not implemented yet
private:
    std::size_t m_size = 0;
    ListNode<T>* m_first = nullptr;
    ListNode<T>* m_last = nullptr;
};

#endif // #define BUW_LIST_HPP

```

Die Listenenden werden intern 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. Legen Sie für dieses Aufgabenblatt wieder ein `cmake` basiertes Repository auf github an. Testen Sie alle Methoden und Funktionen die sie implementieren. **[10 Punkte]**

Aufgabe 4.3

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", "[modifiers]")
{
    List<int> list;
    list.push_front(42);
    REQUIRE(42 == list.front());
}
```

[15 Punkte]

Aufgabe 4.4

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", "[modifiers]")
{
    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 4.5

Die Verwendung von Iteratoren kennen Sie bereits aus dem letzten Aufgabenblatt. 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 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 = nullptr;
};

```

[15 Punkte]

Aufgabe 4.6

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",
          "[iterators]")
{
    List<int> list;

```

```

    auto b = list.begin();
    auto e = list.end();
    REQUIRE(b == e);
}

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

// TEST_CASE ...

```

[5 Punkte]

Aufgabe 4.7

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 4.8

Implementieren Sie nun den Copy-Konstruktor. Überlegen und beschreiben Sie, wie der Compiler den Copy-Konstruktor erzeugt und was bei dessen Ausführung geschieht. Implementieren Sie nun selbst 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);
}

```

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

Aufgabe 4.9

Schreiben Sie die Methode `insert` zum Einfügen eines Objekts vor einer angegebenen Position. Als Argumente dieser Methode werden die Position als Iterator, und das einzufügende Objekt übergeben. Möglicherweise müssen Sie den `ListIterator` anpassen.
[10 Punkte]

Aufgabe 4.10

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 eine neue Liste mit umgekehrter Reihenfolge zurückgibt. Testen Sie diese Methode ausreichend!
[10 Punkte]

Aufgabe 4.11

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

Aufgabe 4.12

Implementieren Sie den Zuweisungsoperator (wie in der Vorlesung am 31. Mai vorgestellt).
[5 Punkte, optional]

Aufgabe 4.13

Implementieren Sie den Move constructor (wie in der Vorlesung am 31. Mai vorgestellt). Verwenden Sie folgende Tests.


```
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());
}
```

[5 Punkte]

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