Programmiersprachen
Sommersemester 2019

Aufgabensammlung 4

Die Aufgaben werden am **Mittwoch**, dem **05. Juni** oder dem **12. Juni** bewertet.

Bitte **melden Sie sich im Moodle** zu einem der beiden Termine zu einem freien Zeit-Slot an. Diese Aufgabensammlung beschäftigt sich mit Pointern, Containern und der Standard Template Library (STL). Es gelten die Ausführungshinweise des vorherigen Aufgabenblattes (github-Repository, `const`-Korrektheit, Initialisierungslisten, 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 adrian.kreskowski@uni-weimar.de.

Aufgabe 4.1

Erstellen Sie unter Ihrem eigenen *github*-Account ein neues Repository mit dem Namen *programmiersprachen-aufgabenblatt-4*. Clonen Sie das erstellte Repository. Erstellen Sie darin die gleiche Struktur wie in den letzten Repositories. Stellen Sie gegebenenfalls in Ihrer CMakeLists.txt-Datei im Root-Verzeichnis des Repositories die Linker-Flags von –std=c++11 auf –std=c++14 um.

Aufgabe 4.2

Machen Sie sich zuerst mit der Struktur einer doppelt-verketteten Liste vertraut (siehe Vorlesung STL-1, S.44). Bevor Sie mit dieser Aufgabe fortfahren, sollte Ihnen klar sein in welchem Zusammenhang die Liste, Listenknoten und Listenelemente stehen. Weiterhin sollte Ihnen die Bedeutung von Iteratoren im Zusammenhang der Liste klar sein.

Implementieren Sie nun selbst eine doppelt verkettete Liste namens `List` als Template-Klasse. Verwenden Sie die hier vorgegebene Header-Datei `List.hpp`. Für maximalen Lerneffekt empfehlen wir, die Klassenstruktur abzuschreiben und sich dabei den Aufbau der Klassen und Datentransferobjekte (DTO) durch den Kopf gehen zu lassen. Alternativ finden Sie ein vorgefertigtes Skelett der `List.hpp` im Moodle zusammen mit der Aufgabe.

Das DTO `ListNode` enthält Zeiger auf das vorherige und das nächste Element, sowie ein Objekt vom Werte-Typen `T`.

Vergleichen Sie die Funktionsweise aller Methoden die Sie implementieren mit den entsprechenden Methoden der **STL-Liste** um sicherzustellen, dass die Funktionalität und das Interface mit dieser kompatibel sind.

In dieser Abnahme beschränken wir uns auf die Teile des Interfaces, welches **keine** const-Iteratoren zurückgibt. Fragen Sie bei Bedarf nach, wenn Funktionalität von Operatoren oder anderen Methoden/Funktionen nicht klar ist.

Vergessen Sie während Ihrer Implementierung die Initialisierung von **Membervariablen** der einzelnen **Klassen** mittels **Initialisierungslisten** nicht!

Analysieren Sie zunächst das Skelett der Template-Klasse im **Anhang** des Blattes und fügen Sie zwischen den Blockkommentaren /* ... */ eine Beschreibung der Funktionsweise der jeweiligen Methoden und freien Funktionen ein.
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;
```

Beachten Sie, dass Listenenden intern durch einen Nullzeiger `nullptr` gekennzeichnet werden.

Die Methode `size()` gibt die Länge der Liste zurück und soll unabhängig von der Anzahl der Listenelemente arbeiten.

Neben der Header-Datei `List.hpp` brauchen Sie nur eine weitere Datei, die Sie `TestList.cpp` nennen können. Diese soll ihre Tests enthalten. Benutzen Sie am besten die mit dem Aufgabenblatt online gestellten Dateien.

Die Definition der Listen-Methoden können Sie für dieses Aufgabenblatt direkt in die **Header**-Datei `Lists.hpp` schreiben, um Linker-Fehler im Zusammenhang mit Template-Instanziierung zu vermeiden. Legen Sie für dieses Aufgabenblatt wieder ein cmake basiertes Repository auf github an. Testen Sie alle Methoden und Funktionen die sie implementieren.

Hinweis: Für **jede Methode und jede Funktion**, die Sie in der Bearbeitung des Aufgabenblattes 4 schreiben gilt: Implementierungen, die **nicht mittels Unit-Tests getestet** wurden, gelten als **falsch implementiert**!

[14 Punkte]

Aufgabe 4.3

Implementieren Sie nun die Methoden `push_front`, `push_back`, `pop_front`, `pop_back`, `front` und `back`. Testen Sie alle Methoden mittels mehrerer Unit-Tests für die verschiedenen Fälle (leere Liste, 1 Element in der Liste, mehrere Elemente in der Liste).
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());
}
```

Hinweis: Kompilieren Sie Ihr Projekt während der Entwicklung im Debug-Mode, (in ccmake oder cmake-gui einstellen), sodass Sie mittels assert fehler zur Laufzeit feststellen können. Gute Kandidaten für `assert` sind Tests, ob die Liste vor den `pop_*()`, `front()` oder `back()` Methoden nicht leer ist. `assert(!empty)`

Hinweis: Vergessen Sie nicht, dass sie dynamische Ressourcen, welche Sie mittels `new` anlegen auch an geeigneten Stellen mit `delete` freigeben müssen um Speicherlecks zu vermeiden!

[15 Punkte]

Aufgabe 4.4

Schreiben Sie die Methode `clear`, welche alle Elemente der Liste löscht. 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** mittels `clear()`.

[5 Punkte]

Aufgabe 4.5

Die Verwendung von Iteratoren kennen Sie bereits aus dem letzten Aufgabenblatt. In dieser Aufgabe sollen Sie selbst einen Iterator implementieren. Fügen Sie dazu folgendes Template in die Datei `List.hpp` über der Definition der Klasse `List` ein.
Implementieren Sie alle Methoden des ListIterators, die als `// not implemented yet` markiert sind. Um die Lesbarkeit zu verbessern haben wir einige Typen-Aliase mittels `using`-clause verwendet. (http://en.cppreference.com/w/cpp/language/type_alias) Dies bedeutet in der Essenz, dass die Typen auf der Linken Seite der using-clause als alternative Namen fürr die Typen auf der rechten Seite benutzt werden können.

```
template <typename T>
struct ListIterator {
  using Self              = ListIterator<T>;
  using value_type        = T;
  using pointer           = T*;
  using reference         = T&;
  using difference_type   = ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;


  /* DESCRIPTION  operator*() */
```

```
T&  operator*()  const {
  //not implemented yet
  return {};
}

/* DESCRIPTION  operator->() */
T* operator->() const {
  //not implemented yet
  return nullptr;
}

/* ... */
ListIterator<T>& operator++() {
  //not implemented yet
  return {};
} //PREINCREMENT

/* ... */
ListIterator<T> operator++(int) {
  //not implemented yet
  return {};
} //POSTINCREMENT (signature distinguishes)

/* ... */
bool operator==(ListIterator<T> const& x) const {
  ////not implemented yet
}

/* ... */
bool operator!=(ListIterator<T> const& x) const {
  ////not implemented yet
}

/* ... */
ListIterator<T> next() const {
  if (nullptr != node) {
    return ListIterator{node->next};
  } else {
    return ListIterator{nullptr};
  }
}
```

```
  ListNode <T>* node = nullptr;
};
```

Hinweis: Ihre operator-> Implementierung kann nur mit zusammengesetzten Datentypen getestet werden, da der Pfeiloperator dereferenziert und einen Member nach der Dereferenzierung erwartet. Testen Sie diese Funktion z.B. mit einer Liste von Circle-Objekten:

```
List<Circle> circle_list;
circle_list.push_back(...);

auto c_it = circle_list.begin();

std::cout << "Der Radius des 1. Circles in der Liste ist: "
          << c_it->get_radius() << std::endl;
```

[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 access 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 **Member**-Funktionen:

```
template<typename T>
bool operator==(List<T> const& rhs)
{
  // do not use .begin() and .end!
  // use first_ and rhs.first_ and check for nullpointers!
  // not implemented yet
}

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

Hinweis: Die (Un-)-Gleichheitskriterien von Listen können Sie bei Bedarf auf en.cppreference nachlesen:
(`https://en.cppreference.com/w/cpp/container/list/operator_cmp` unter 1-2))

[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);
}
```

Hinweis 1: Für den Copy-Constructor können Sie die `push_*` Methoden verwenden.
Hinweis 2: Beachten Sie den Unterschied zwischen *Flacher Kopie (Shallow Copy)* und *Tiefer Kopie (Deep Copy)*
Hinweis 3: Benutzen Sie direkt die Member `first_` und `last_` anstatt der Methoden begin() und end() im Kopiekonstruktor um die Implementierung von ListConstIteratoren zu vermeiden.

[5 Punkte]

Aufgabe 4.9

Schreiben Sie die Methode `insert` zum Einfügen eines Objekts vor einer angegeben Position. Als Argumente dieser Methode werden die Position vor dem das neue Element eingefügt werden soll als Iterator übergeben. Die Funktion gibt zusätzlich einen Iterator auf das eingefügte Element in der Liste zurück.

Überlegen Sie sich selbst wie das Interface für die Methode `insert` auszusehen hat und fügen Sie diese entsprechend in Ihre Klassendefinition ein.

Hinweis: Iterieren Sie **NICHT** über die komplette Liste, sondern benutzen Sie den übergebenen Iterator um das neue Element einzuhängen!

[10 Punkte]

Aufgabe 4.10

Implementieren Sie die Methode `reverse`, welche die Reihenfolge der Liste durch umkehrt. Realisieren Sie dies durch austauschen der next and prev pointer der Knoten. umkehrt. Implementieren Sie weiterhin 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 Instanz ihrer eigenen Klasse `List<int>` in einen `std::vector<int>` zu kopieren.

[5 Punkte, optional]

Aufgabe 4.12

Implementieren Sie den Zuweisungsoperator (wie in der Vorlesung vorgestellt).

[5 Punkte, optional]

Aufgabe 4.13

Implementieren Sie den Move-Constructor (wie in der Vorlesung 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]

Aufgabe 4.14

Informieren Sie sich auf *cppreference.com* über die Template-Klasse `std::initializer_list` (http://en.cppreference.com/w/cpp/utility/initializer_list), welche im Header `<initializer_list>` definiert ist.

Implementieren Sie anschließend einen Konstruktor für ihre List-Klasse, welcher ein Objekt vom Typ `std::initializer_list<T>` übergeben bekommt und deren Inhalte mittels range-based for-loop und Ihrem `push_back`-Interface in Ihre Liste einfügt.

Wenn Sie erfolgreich waren sollte sich ihre eigene Liste mit brace-enclosed initialiser list wie folgt kompilieren lassen:

```
List<int> int_list{9, 5, 38, 100};
```

Testen Sie Ihren neuen Konstruktor!

Implementieren Sie nun den operator+ für Ihre Liste. Dieser Operator soll zwei Listen des gleichen Typs entgegennehmen und zu einer dritten Liste konkatenieren, sodass

```
List<int>{1, 2} + List<int>{5, 6}
```

ein Objekt äquivalent zu

```
List<int>{1, 2, 5, 6}
```

zurückgibt.

Finden Sie abschliessend heraus und weisen Sie nach, wie oft der Move-Konstruktor in der folgenden Zeile aufgerufen wird:

```
auto l = List<int>{1, 2, 3, 4, 5} + List<int>{6, 7, 8, 9};
```

[5 Punkte, optional]

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

Anhang

```
#ifndef BUW_LIST_HPP
#define BUW_LIST_HPP
#include <cassert>
#include <cstddef>  //ptrdiff_t
#include <iterator> //std::bidirectional_iterator_tag
#include <iostream>

#include <initializer_list>

template <typename T>
class List;

template <typename T>
struct ListNode {
  T         value = T ();
  ListNode* prev = nullptr;
```

```
  ListNode* next = nullptr;
};

template <typename T>
struct ListIterator {
  using Self              = ListIterator<T>;
  using value_type        = T;
  using pointer           = T*;
  using reference         = T&;
  using difference_type   = ptrdiff_t;
  using iterator_category = std::bidirectional_iterator_tag;


  /* DESCRIPTION  operator*() */
  T&  operator*()  const {
    //not implemented yet
    return {};
  }

  /* DESCRIPTION  operator->() */
  T* operator->() const {
    //not implemented yet
    return nullptr;
  }

  /* ... */
  ListIterator<T>& operator++() {
    //not implemented yet
    return {};
  } //PREINCREMENT

  /* ... */
  ListIterator<T> operator++(int) {
    //not implemented yet
    return {};
  } //POSTINCREMENT (signature distinguishes)

  /* ... */
  bool operator==(ListIterator<T> const& x) const {
    ////not implemented yet
  }
```

```
  /* ... */
  bool operator!=(ListIterator<T> const& x) const {
    ////not implemented yet
  }

  /* ... */
  ListIterator<T> next() const {
    if (nullptr != node) {
      return ListIterator{node->next};
    } else {
      return ListIterator{nullptr};
    }
  }


  ListNode <T>* node = nullptr;
};



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

    // not implemented yet
    // do not forget about the initialiser list !
    /* ... */
    List() {}

    /* ... */
    //TODO: Copy-Konstruktor using Deep-Copy semantics (Aufgabe 4.8)

    /* ... */
    //TODO: Move-Konstruktor (Aufgabe 4.13)
```

```
//TODO: Initializer-List Konstruktor (4.14)
/* ... */
List(std::initializer_list<T> ini_list) {
  //not implemented yet
}

/* ... */
//TODO: Assignment operator (Aufgabe 4.12)

/* ... */
//TODO: operator== (Aufgabe 4.7)

/* ... */
//TODO: operator!= (Aufgabe 4.7)

/* ... */
~List() {
  //TO IMPLEMENT PROPERLY
}

/* ... */
ListIterator<T> begin() {
  ////not implemented yet
  return ListIterator<T>{};
}

/* ... */
ListIterator<T> end() {

  ////not implemented yet
  return ListIterator<T>{};
}

/* ... */
void clear() {
  ////not implemented yet
}

/* ... */
//TODO: member function insert
```

```
/* ... */
//TODO: member function reverse

/* ... */
void push_front(T const& element) {
  //not implemented yet
}

/* ... */
void push_back(T const& element) {
  //not implemented yet
}

/* ... */
void pop_front() {
  assert(!empty());
  //not implemented yet
}

/* ... */
void pop_back() {
  assert(!empty());
//not implemented yet
}

/* ... */
T& front() {
  assert(!empty());
  //not implemented yet

  return T(); //<- obviously wrong because of
        // returned reference to tmp-Object
}

/* ... */
T& back() {
  assert(!empty());

  //not implemented yet

  return T(); //<- obviously wrong because of
```

```
            // returned reference to tmp-Object
    }

    /* ... */
    bool empty() const {
      //not implemented yet
      return false;
    };

    /* ... */
    std::size_t size() const {
      //not implemented yet
      return std::numeric_limits<std::size_t>::max();
  };


  private:
    std::size_t size_;
    ListNode<T>* first_;
    ListNode<T>* last_;
};

/* ... */
//TODO: Freie Funktion reverse

/* ... */
//TODO: Freie Funktion operator+ (4.14)

#endif // # define BUW_LIST_HPP
```