Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Fortgeschrittene Themen in C++

Aufgabe 14 [F] Generische Funktionen (Templates)

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/templates`.

Aufgabe 14.1 Templatefunktionen implementieren

Implementiere die folgende Funktion, die das Maximum von zwei Variablen liefert:

```
template<typename T>
const T &maximum(const T &t1, const T &t2);
```

Durch die Verwendung von Templates soll die Funktion mit verschiedenen Datentypen funktionieren. Teste deine Implementation.

In der Vorlesung haben wir gesehen, dass jede Verwendung von `t1` und `t2` in `maximum` eine Schnittstelle induziert, die der Typ `T` bereitstellen muss. Das bedeutet, dass `T` alle Konstruktoren, Methoden und Operatoren zur Verfügung stellen muss, die in `maximum` genutzt werden.

Wie sieht diese Schnittstelle in diesem Fall aus? Welche Gründe gibt es, den Rückgabewert der Funktion `maximum` als konstante Referenz festzulegen?

Hinweise

- In den meisten Fällen kann anstelle von `typename` auch `class` in der Template-Deklaration verwendet werden.
- In der Regel muss die Definition von Template-Funktionen und -Methoden im Header erfolgen, damit der Compiler das auszufüllende Template „textuell“ vorliegen hat. Dies ist der sicherere Weg und immer dann notwendig, wenn man nicht weiß, welche Spezialisierungen des Templates in Zukunft benötigt werden (bspw. bei der Entwicklung von Bibliotheken wie der STL).

  Alternativ kann man die Definition der Methoden einer Templateklasse auch in einer `cpp`-Datei angeben. In diesem Fall muss man dafür sorgen, dass die benötigten Spezialisierungen auch in der `cpp`-Datei eingefordert werden (bspw. durch ein `typedef`). Weitere Spezialisierungen in anderen `cpp`-Dateien sind dann aber nicht mehr möglich. Diese Option ist eher in Sonderanwendungsfällen sinnvoll und hat den Vorteil, dass man die Implementierung weiterhin vom Header getrennt hält.

Aufgabe 14.2 Explizite Angabe der Typparameter

Lege nun zwei Variablen vom Typ `int` und `short` an, und versuche, mittels `maximum()` das Maximum zu bestimmen. Der Compiler wird mit der Fehlermeldung **no matching function for call...** abbrechen, da er nicht weiß, ob `int` oder `short` der Template-Parameter sein soll. Gib deshalb den Template-Parameter mittels `maximum<int>()` beim Aufruf von `maximum()` explizit an. Die übergebenen Parameter werden dabei vom Compiler automatisch in den gewünschten Typ umgewandelt.

Aufgabe 14.3 Induzierte Schnittstelle implementieren

Erstelle eine Klasse `C`, die eine Zahl als Attribut beinhaltet. Implementiere einen passenden Konstruktor sowie einen Getter für diese Zahl. Nun wollen wir unsere Funktion `maximum()` verwenden, um zu entscheiden, welches von zwei `C`-Objekten die größere Zahl beinhaltet. Überlege dir, was zu tun ist, und implementiere es.

Hinweise

- Die Klasse `C` muss mindestens die durch `maximum` induzierte Schnittstelle implementieren.

Aufgabe 15 [F] Generische Vektor-Implementation (Templates)

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/generic_vector`. Erinnere dich an die Klasse `Vector3` aus dem ersten Praktikumstag (**??**). Diese hat den Datentyp `double` für die einzelnen Komponenten verwendet. Schreibe die Klasse so um, dass der Datentyp der Komponenten durch einen Template-Parameter angegeben werden kann. Füge dafür der Klasse `Vector3` einen Template-Parameter hinzu und ersetze jedes Aufkommen von `double` mit dem Template-Parameter. Vergiss nicht, die Implementation in den Header zu verschieben, da der Compiler die Definition einer Klasse kennen muss, um beim Einsetzen des Template-Parameters den richtigen Code zu generieren.

Verbessere außerdem die Effizienz und Sauberkeit der `Vector3`-Klasse, in dem du die Parameterübergabe in den entsprechenden Methoden auf `const` Referenzen umstellst und alle Getter als `const` deklarierst.

Du weißt bereits, dass alle `template`-Funktionen und -Methoden im Header enthalten sein müssen. Um den Code trotzdem zu strukturieren, hat es sich eingebürgert, dass man die Klassendefinition in der `hpp`-Datei hält, ohne die Methoden zu implementieren. Im Anschluss wird eine `tpp`-Datei inkludiert, die die Implementierung der Methoden und Funktionen enthält. Der Aufbau der Datei `Vector3.hpp` wäre also wie folgt:

```
#ifndef VECTOR3_HPP_
#define VECTOR3_HPP_

/**
 * Don't forget documentation!
 */
template<typename T>
class Vector3 {
public:
    // Method declarations

    // You need to use a different template type if you want to declare the
    // overloaded output operator as friend.
    // Otherwise, you can introduce getters for the vector attributes.
    // Then there is no need to use a friend declaration.
    template<typename X>
    friend std::ostream& operator<<(std::ostream& out, const Vector3<X> rhs);

private:
    // Attributes
};

// function declarations only

#include "Vector3.tpp" // contains method and function definitions

#endif /* VECTOR3_HPP_ */
```

Hinweise

- Die Datei `Vector3.tpp` ist nicht vorgegeben, du musst diese selbst erstellen!
- Auch wenn bei reinen Template-Klassen die `cpp`-Datei leer bleibt, ist es sinnvoll, eine solche anzulegen. Dadurch wird das Template garantiert auf Syntaxfehler überprüft. Der Inhalt der `cpp`-Datei ist in dieser Aufgabe schlicht `#include "Vector3.hpp"`.

Aufgabe 16 [F] Generische Verkettete Liste (Templates)

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/generic_linked_list`.

Aufgabe 16.1

Schreibe die Klassen `List`, `ListItem` und `ListIterator` aus dem zweiten Praktikumstag so um, dass man den Typen der in der Liste gespeicherten Elemente über einen Template-Parameter angeben kann.

Dazu müssen einige Änderungen gemacht werden. Zum einen sollte der Inhalt eines Elements beim Erstellen nicht als Wert sondern als `const` Referenz übergeben werden. Zum anderen sollten die Methoden zum Löschen von Elementen `void` zurückgeben, und nicht mehr das jeweilige gelöschte Element. Der Grund dafür ist, dass in diesem Fall eine temporäre Kopie des Elements gemacht werden müsste, ohne dass es der Benutzer beeinflussen kann. Je nach Elementtyp können solche Kopien problematisch und unerwünscht sein.

Hinweise

- Arbeite die Klassen nacheinander ab, beginnend bei `ListItem`.
- Stelle sicher, dass man eine Klasse fehlerfrei kompilieren kann, bevor du zur nächsten übergehst.
- Denke daran, dass du auch hier die Implementation in eigene `*.tpp`-Dateien verschieben musst.

Aufgabe 16.2

Überlade den `operator<<`, sodass Listen direkt über ein `std::ostream` wie z.B. `std::cout` ausgegeben werden können.

Aufgabe 16.3

Teste deine Implementierung. Probiere auch Folgendes aus und beobachte die Ausgabe.

```
List<List<int> > list;
list.appendElement(List<int>());
list.getFirst().appendElement(1);
list.getFirst().appendElement(2);
list.appendElement(List<int>());
list.getLast().appendElement(3);
list.appendElement(List<int>());
list.getLast().appendElement(4);
list.getLast().appendElement(5);
std::cout << list << std::endl;
```

Hinweise

- In der ersten Zeile ist absichtlich ein Leerzeichen zwischen den beiden schließenden spitzen Klammern. Bis hin zu C++11 konnte der C++-Compiler nicht erkennen, ob es sich bei >> um den Operator oder um geschachtelte Templates handelt. Seit C++11 ist es nicht mehr nötig, ein Leerzeichen zwischen die beiden schließenden spitzen Klammern einzufügen.

Aufgabe 17 [F] Funktionales Programmieren (Funktionszeiger, Funktoren)

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/functional_programming`.

In dieser Aufgabe werden Funktionen aus der funktionalen Programmierung vorgestellt. Diese sind `map`, `filter` und `reduce`. Der Ablauf ist wie folgt:

- In Aufgabe 17.1 werden erst einmal die Funktionsweisen der zu implementierenden Funktionen `map`, `filter` und `reduce` vorgestellt.
- In Aufgabe 17.2 wirst du diese Funktionen und zusätzliche Hilfsfunktionen implementieren.
- In Aufgabe 17.3 wirst du deine Hilfsfunktionen als *Funktoren* implementieren und `map`, `filter` und `reduce` entsprechend anpassen.
- In Aufgabe 17.4 wirst du den Code auf Templates umstellen, damit Funktionen und Funktoren austauschbar verwendet werden können.

Aufgabe 17.1 Erklärung `map`, `filter` und `reduce`

Arbeitet man auf iterierbaren Sequenzen, ist dies fast immer mit Schleifen über die Sequenz verbunden. Die drei Funktionen `map`, `filter` und `reduce` vereinfachen uns hierbei die Arbeit. Hierzu ein Beispiel: Haben wir einen Vektor des Typs `double` und wollen jedes Element quadrieren, endet dies meist in dem folgenden Programmcode:

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

for (std::vector<double>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
    *it = square(*it); // squaring element
}

// numbers now [ 1, 4, 9, 16, 25 ]
```

map

Die Idee von der Funktion `map` ist es, genau dies zu vereinfachen. Die Funktion erhält folgende Parameter:

- Die Start- und Enditeratoren der zu modifizierenden Sequenz
- Einen Iterator, der auf eine Sequenz zeigt, in der die veränderten Elemente gespeichert werden sollen
- Einen Funktionszeiger, der auf eine Funktion zeigt, die für jedes Element der iterierbaren Sequenz aufgerufen werden soll

Ein Beispiel siehst du in folgendem Listing:

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

map(numbers.begin(), numbers.end(), numbers.begin(), square);
// numbers now [ 1, 4, 9, 16, 25 ]
```

filter

Die Funktion `filter` funktioniert analog, indem sie einen Zeiger auf eine Funktion erhält, die einen Listenelementtyp erwartet und ein Booleschen Wert (`bool`) zurückgibt. Auf alle Elemente wird diese Funktion aufgerufen und alle Elemente, für die die Funktion `true` zurückgibt, werden in die Ausgabesequenz kopiert. Der Rest wird entfernt.

```
// #include <iterator>

std::vector<double> numbers = { 1, 2, 3, 4, 5 };
std::vector<double> filteredNumbers;

filter_funcpointer(numbers.begin(), numbers.end(),
                   std::back_inserter(filteredNumbers), is_odd);

// filteredNumbers now [ 1, 3, 5 ]
```

Die Besonderheit hier ist, dass wir eine zweite Liste (`filteredNumbers`) benötigen, um das Ergebnis zu speichern, da die Ergebnisliste in der Regel kürzer sein wird als die Eingabeliste. Eine andere Lösung wäre, den letzten Stand des Ausgabeiterators zurückzugeben und die Liste entsprechend einzukürzen.

Zusätzlich zur Ausgabeliste verwenden wir hier einen `std::back_insert_iterator`[1], der die ihm geordnete Liste immer dann vergrößert, wenn ein neues Element in den Iterator hineingeschrieben wird. Die Hilfsfunktion `std::back_inserter`[2] vereinfacht die Erzeugung des Iterators. Sowohl die Klasse `std::back_insert_iterator` als auch die Hilfsfunktion `std::back_inserter` befinden sich im Header `iterator`.

reduce

Die Aufgabe der Funktion `reduce` ist es, eine Sequenz zu einem einzelnen Element zusammenzuschrumpfen. Hierbei wird der Ausgabeiterator gegen einen Startwert ausgetauscht. Hier ein Beispiel, bei dem die Summe über die Elemente in `numbers` gebildet wird.

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

std::cout << reduce(numbers.begin(), numbers.end(), 0.0, sum) << std::endl; // 15
```

Aufgabe 17.2 Programmieren der Funktionen

Du wirst nun die drei Funktionen `map`, `filter` und `reduce` nachprogrammieren. Hierbei geht es erstmal darum, ein funktionierendes Gerüst zu erstellen, anstatt perfekt generische Algorithmen zu erhalten.

Aufgabe 17.2.1 `map`

Schreibe eine Funktion `map` die folgende Signatur besitzt.

```
template<typename InIt, typename OutIt>
OutIt map(InIt first, InIt last, OutIt out_first, double(*func)(double d));
```

Hierbei ist der letzte Parameter der Funktionszeiger. Die Klammern um `*func` sind notwendig, damit der Compiler den übergebenen Parameter als Funktionszeiger einer Funktion mit Rückgabewert `double` interpretiert und nicht als Funktion mit Rückgabewert `double *`[3]. Diese Funktion hat zusätzlich noch ein `double d` als Parameter.

Du kannst dich bei der Implementierung von `map` an dem Schleifengerüst zu Anfang von Aufgabe 17.1 orientieren.

Aufgabe 17.2.2 `filter`

Die von dir zu schreibende Funktion `filter` soll der folgenden Signatur folgen.

```
template<typename InIt, typename OutIt>
OutIt filter(InIt first, InIt last, OutIt out_first, bool(*pred)(int i));
```

[1] `http://en.cppreference.com/w/cpp/iterator/back_insert_iterator`
[2] `http://en.cppreference.com/w/cpp/iterator/back_inserter`
[3] Welche folgende Signatur hätte: `double *(*func)(double d)`

Die Implementierung von `filter` wird sehr ähnlich zur Implementierung von `map` aussehen. Der Hauptunterschied ist, dass `pred` in einer `if`-Bedingung eingesetzt werden muss, um zu entscheiden, ob der aktuelle Wert in den Ausgabeiterator (`out_first`) geschrieben werden soll.

Aufgabe 17.2.3 `reduce`

Erstelle eine Funktion `reduce`, die der folgenden Signatur folgt.

```
template<typename InIt, typename RetT, typename RHS>
RetT reduce(InIt first, InIt last, RetT initialVal, RetT(*func)(RetT i, RHS j));
```

Hierbei muss ein passender initialer Wert übergeben werden, der mit dem Rückgabewert und dem ersten Argument der übergebenen Funktion zusammenpasst (`typename RetT`). Der zweite Parameter der Funktion kann sich im Typ sogar unterscheiden (`typename RHS`).

Aufgabe 17.2.4 Hilfsfunktionen implementieren

Implementiere in dieser Aufgabe drei Hilfsfunktionen, die den Anforderungen der jeweiligen Signaturen der Funktionszeiger in den Funktionen `map`, `filter` und `reduce` folgen. Du kannst dir dabei gerne eigene Funktionen ausdenken oder dich an die Funktionen in den Beispielen halten (bspw. `square` für `map`, `isOdd` für `filter`, `sum` für `reduce`).

Teste anschließend deine Implementierungen mithilfe deiner Hilfsfunktionen.

Aufgabe 17.3 Funktoren

Es gibt außerdem noch die Möglichkeit, Funktionen in einem Funktionsobjekt (*Funktor*) zu kapseln. Dabei überlädt man den Operator `operator()` der Funktor-Klasse, welcher eine bestimmte Funktion ausführt. Schaut man sich in unserem Beispiel die Funktion `square` mit der Definition `double square(double i);` an, würde der Funktor folgendermaßen aussehen:

```
class Square {
public:
   double operator() (double i) { return i*i; }
};
```

Die Funktion `map` würde wie folgt umgeschrieben werden müssen, um den Funktor zu akzeptieren:

```
template<typename InIt, typename OutIt>
OutIt map(InIt first, InIt last, OutIt out_first, Square s);
```

Erstelle für jede deiner Hilfsfunktionen eine Funktor-Klasse und füge neue Implementierungen für `map`, `filter` und `reduce` hinzu, die mit den Funktoren kompatibel sind. Vergleiche die Ausgabe der Funktionszeiger- und Funktoren-basierten Implementierungen.

Aufgabe 17.4 Verwendung von Templates

Die derzeitige Implementierung funktioniert entweder mit Funktoren einer bestimmten Klasse oder mit Funktionszeigern, die einem bestimmten Typen angehören, der durch die Signatur der Funktion festgelegt ist. Diese Verdoppelung des Codes ist unschön und Um das Problem zu lösen, kann man die Funktionszeiger-/Funktorvariable durch einen Templateparameter ersetzen. Damit ist der Parametertyp flexibel; der Nachteil ist, dass nur noch aus der eigentlichen Implementierung der Funktion hervorgeht, was der Typ des Templateparameters anbieten muss (*Induzierte Schnittstelle*).

Erstelle Varianten der Funktionen `map`, `filter` und `reduce`, deinen weiteren Templateparameter angeben, der flexibel mit Funktionszeigern oder Funktoren belegt werden kann.

```
template <typename InIt, typename OutIt, typename ...>
```

Vergleiche die Ausgabe deiner Template-basierten Lösung mit den Ergebnissen der Funktionszeiger- und Funktor-basierten Lösungen.

Nachwort zu dieser Aufgabe

Für produktive C++-Programme bietet die Standardbibliothek fertige Funktionen und Klassen, um die gerade erlernten Prinzipien dieser Aufgabe zu realisieren, z.B. `std::function<...>`[4] und `std::bind()`[5]. Diese können mit einer beliebigen Anzahl von Parametern umgehen und beinhalten viele weitere Features.

[4] `http://en.cppreference.com/w/cpp/utility/functional/function`
[5] `http://en.cppreference.com/w/cpp/utility/functional/bind`

Aufgabe 18 [F] Standard-Container

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/standard_container`. In dieser Aufgabe werden wir den Umgang mit den Containern `std::vector` und `std::list` aus der Standard Template Library üben. Es ist sinnvoll, wenn du während der Übung eine C++-Referenz zum Nachschlagen bereithältst, z.B. `http://www.cplusplus.com/`. Schaue dir auch die Vorlesungsfolien genau an, da diese nützliche Codebeispiele enthalten.

Die Klasse `std::list` stellt eine verkettete Liste dar, bei der man an beliebiger Stelle Elemente effizient löschen und hinzufügen kann. `std::vector` stellt ähnliche Funktionen bereit, allerdings liegen hier die Elemente in einem einzigen, zusammenhängenden Speicherbereich, der neu alloziert und kopiert werden muss, wenn seine aktuelle Kapazität überschritten wird. Auch müssen viele Elemente verschoben werden, wenn der Vektor in der Mitte oder am Anfang modifiziert wird. Der große Vorteil von `std::vector` ist der *wahlfreie Zugriff*, d.h. man kann auf beliebige Elemente mit konstantem Aufwand zugreifen.

a) Schreibe zunächst eine Funktion `template<typename T> void print(const T &t)`, die beliebige Standardcontainer auf die Konsole ausgeben kann, die Integer speichern und Iteratoren unterstützen. Nutze dazu die Funktion `copy()` sowie die Klasse `std::ostream_iterator<int>`, um den entsprechenden `OutputIterator` zu erzeugen.

b) Lege ein `int`-Array an und initialisiere es mit den Zahlen 1 bis 5. Lege nun einen `std::vector<int>` an und initialisiere ihn mit den Zahlen aus dem Array.

c) Lege eine Liste `std::list<int>` an und initialisiere diese mit dem zweiten bis vierten Element des Vektors. **Tipp**: Du kannst auf Iteratoren eines Vektors (genauso wie auf Zeiger) Zahlen addieren, um diese zu verschieben.

d) Füge mittels `std::list<T>::insert()` das letzte Element des Vektors an den Anfang der Liste hinzu.

e) Lösche alle Elemente des Vektors mit einem einzigen Methodenaufruf.

f) Mittels `remove_copy_if()` kann man Elemente aus einem Container in einen anderen kopieren und dabei bestimmte Elemente löschen lassen. Nutze diese Funktion, um alle Elemente, die kleiner sind als 4, aus der Liste in den Vektor zu kopieren. Beachte, dass `remove_copy_if()` keine neuen Elemente an den Container anhängt, sondern lediglich Elemente von der einen Stelle zur anderen elementweise durch Erhöhen des `OutputIterator` kopiert.

   Deshalb kannst du `vec.end()` **nicht** als `OutputIterator` nehmen, da dieser "hinter" das letzte Element zeigt und weder dereferenziert noch inkrementiert werden darf. Nutze stattdessen die Methode `back_inserter()`, um einen Iterator zu erzeugen, der neue Elemente an den Vektor anhängen kann.

Aufgabe 19 [F] UnitTest++ (optional)

Diese Aufgabe dient der Vertiefung deines Wissens in C++ und ist nicht notwendig, um die Klausur zu bestehen.

Ein Lösungsvorschlag für diese Aufgabe liegt im Ordner `./exercises/solutions/generic_linked_list_tests`.

Bisher hast du vermutlich durch „scharfes Draufschauen“ und Debuggen sichergestellt, dass deine Implementierungen richtig funktionieren. In dieser Aufgabe wollen wir das Testframework *UnitTest++*[6] benutzen um das Testen des Codes zu automatisieren.

In der virtuellen Maschine des Praktikums ist UnitTest++ bereits vorinstalliert. Falls du dein eigenes System nutzen willst, kannst du UnitTest++ wie unter folgender URL beschrieben einrichten: `http://codelite.org/LiteEditor/UnitTestPP`.

Als Grundlage dieser Aufgabe kannst du deine eigene Implementierung der verketteten Liste nutzen (siehe Aufgabe 16) oder die entsprechende Musterlösung aus folgendem Verzeichnis in CodeLite importieren: `./exercises/solutions/generi`

Gehe nun wie folgt vor:

a) Erstelle in deinem Workspace ein neues **UnitTest++-Projekt** (*Rechtsklick* → *New* → *New Project* → *UnitTest++/UnitTest++*).

b) Wähle wie üblich einen **Namen** und den **Compiler** des Projekts aus.

c) Damit du auf die Header des zu testenden Projekts zugreifen kannst, füge zum Include-Pfad des Projekts den Pfad zum Projekt generic_linked_list hinzu (bspw. den relativen Pfad `../generic_linked_list` oder den absoluten Pfad `/home/cppp/Repos/tud-cppp/exercises/solutions/generic_linked_list`): *Rechtsklick* → *Settings* → *Compiler/Include Path*.

d) Falls du Funktionen aus `.cpp`-Dateien testen möchtest, ist es in CodeLite am einfachsten, einen virtuellen Ordner im TestProjekt (also generic_linked_list_tests) zu erstellen (*Rechtsklick* → *New Virtual Folder*) und per *Rechtsklick* → *Add an Existing File* die entsprechenden `.cpp`-Dateien hinzuzufügen.

e) Um sicherzustellen, dass das zu testende Projekt immer vor dem Testprojekt compiliert wird, verwendest du die *Build Order* von CodeLite. Navigiere wie folgt *Rechtsklick auf generic_linked_list_tests* → *Build Order...* und wähle das Projekt *generic_linked_list* aus.

f) Erstelle nun wie im folgenden Codeausschnitt skizziert 4 Unit-Tests mithilfe der Makros `CHECK_EQUAL` und `CHECK_THROW`. Dazu erzeugst du dir für jeden Test ein passendes `List`-Objekt. Hierzu zwei Beispiele:

- `CHECK_EQUAL(3, someFunction());` testet, dass ein Aufruf von `someFunction` den Wert 3 ergibt.
- `CHECK_THROW(f(), std::out_of_range);` testet, dass beim Aufruf der Funktion `f` Exception von Typ `std::out_of_range` geworfen wird.

Hier findest du weitere Informationen zu den von UnitTest++ bereitgestellten Makros.

```
#include <UnitTest++/UnitTest++.h>

#include "List.hpp"

TEST(ListSimpleUsageTest)
{
   // Teste appendElement und deleteFirst für List<int>
  // Wie sind die erwarteten Werte von getSize() und ggf. getFirstElement()?
}
```

[6] Webseite: `https://unittest-cpp.github.io/`

```
TEST(ListAdvancedUsageTest) {

   // Füge an eine List<double> Elemente an und an beliebieger Stelle eine
   // prüfe,ob das n-te Element für jeden Eintrag der Erwartung entpricht
}

TEST(ListIteratorTest)
{
   // Nutze den Iterator um wortweise "Hello World!" einer Liste hinzuzufügen.
}

TEST(ExceptionsTest)
{
   // Teste die 3 Methoden, bei denen Exceptions auftreten können.
}

int main(int, char **)
{
   return UnitTest::RunAllTests();
}
```

g) Achte insbesondere darauf, dass der Include für UnitTest++ wie folgt lautet: `#include <UnitTest++/UnitTest++.h>`. Die standardmäßig von CodeLite generierte Datei enthält hier eine falsche Groß- und Kleinschreibung.

h) Compiliere und linke das Projekt wie gewohnt und führe das Programm aus. Du erhältst im Reiter UnitTest++ eine grafische Ausgabe des Testlaufs und, falls Tests fehlschlagen, eine aussagekräftige Fehlermeldung.