

# Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Übungen für den 2. Tag

### Musterlösungs-/Microcontroller-Projekte in CodeLite importieren

Die angebotenen Musterlösungs-/Microcontroller-Projekte basieren auf Makefiles. Daher ist es wichtig, dass du sie entsprechend importierst: **Workspace -> Add an existing project** und dann zur entsprechenden .project-Datei navigieren.

### Aufgabe 1 Zeiger und Referenzen Grundlagen

In dieser Aufgabe sollst du den Umgang mit Zeigern (*Pointer*) und Referenzen erlernen. Diese erlauben es zum Beispiel Werte zwischen Funktionen auszutauschen ohne eine Kopie der zu übermittelnden Daten zu erzeugen. Anstelle dessen kann ein (vergleichsweise kleiner) Zeiger auf einen Speicherbereich übergeben werden. Alternativ kann auch eine Referenz auf eine Variable übergeben werden, welche intern ähnlich wie ein Zeiger gehandhabt wird.

#### Aufgabe 1.1 Experimente

Experimentiere mit Zeigern, Adressen und Referenzen. Mache dir eine Skizze, die verdeutlicht, wie Variablen und ihre Speicherabbilder zusammenhängen. Als Ausgangsbasis kann folgendes Programmfragment dienen.

```
int intVal = 42;
int *pIntVal = &intVal;
cout << "Wert von intVal " << intVal << endl;
cout << "Wert von &intVal " << &intVal << endl;
cout << "Wert von pIntVal " << pIntVal << endl;
cout << "Wert von *pIntVal " << *pIntVal << endl;
cout << "Wert von &pIntVal " << &pIntVal << endl;
```

#### Aufgabe 1.2 Bedeutung verstehen

Versuche die Bedeutung folgender Ausdrücke zu verstehen. Welche Regelmäßigkeiten stellst du fest?

```
int intVal = 42;
int *pIntVal = &intVal;
*&intVal;
*&pIntVal;
&*pIntVal;
**&pIntVal;
* *&intVal;
&* &pIntVal;
*&*pIntVal;
```

#### Hinweise

- Gehe dabei von rechts nach links vor.

#### Aufgabe 1.3 Gültigkeit

Warum sind folgende Ausdrücke ungültig, sinnlos oder sogar gefährlich?

---

## Übungen für den 2. Tag

---

```
*intVal;  
**pIntVal;  
**&*pIntVal;  
&*intVal;  
&42;
```

### Hinweise

- Finde heraus, welchen Typ der Ausdruck hätte haben müssen.
- Nur tatsächlich angelegte Variablen haben Adressen. Ausdrücke wie `a + b` oder direkt kodierte Zahlenlitterale wie `42` haben keine Adresse-

---

### Aufgabe 1.4 Variablentausch

---

Schreibe eine Funktion `swap`, die zwei `int`-Variablen miteinander vertauscht. Probiere dabei beide möglichen Übergabevarianten (per Referenz, per Pointer) aus. Was würde passieren, wenn man die Variablen stattdessen per Wert übergeben würde?

---

### Aufgabe 1.5 Programmanalyse

---

Sieh dir folgendes Programm an.

```
#include <iostream>  
  
void foo(int &i) {  
    int i2 = i;  
    int &i3 = i;  
  
    std::cout << "i = " << i << std::endl;  
    std::cout << "i2 = " << i2 << std::endl;  
    std::cout << "i3 = " << i3 << std::endl;  
    std::cout << "&i = " << &i << std::endl;  
    std::cout << "&i2 = " << &i2 << std::endl;  
    std::cout << "&i3 = " << &i3 << std::endl;  
}  
  
int main() {  
    int var = 42;  
    std::cout << "&var = " << &var << std::endl;  
    foo(var);  
}
```

Welche Adressen werden übereinstimmen, welche werden sich unterscheiden? Führe das Programm aus. Hast du diese Ausgabe erwartet?

---

### Aufgabe 1.6 Const Correctness

---

In dieser Aufgabe setzt du dich mit der Bedeutung des Schlüsselworts `const` im Kontext von Pointern auseinander. Versuche für jede der Variablen im folgenden Code je eine *Verwendung* zu finden, die

- gültig ist (= fehlerfrei kompiliert) und
- nicht gültig ist (= einen Compiler-Fehler wirft).

Was ist jeweils der Grund? Welche Pointer verhalten sich gleich?

## Übungen für den 2. Tag

```
int i = 1;
int *iP = &i;
const int *ciP = &i;
int const *ciP2 = &i;
int * const icP = &i;
const int * const cicP = &i;
```

### Mehrstufige Pointer

Versuche nun das Gleiche mit den folgenden mehrstufigen Pointern.

```
int **iPP = &iP;
const int * const *cicPP = &iP;
int ** const iPcP = &iP;
```

### Aufgabe 1.7 Übergabewerte

In der letzten Aufgabe direkt zu Pointern geht es darum, das gerade erlangte Verständnis über Pointer und Referenzen zu festigen und zu kontrollieren. In Tabelle 1 sind in der ersten Spalte Funktionen mit verschiedenen Parametern gegeben. In der ersten Zeile findet ihr verschiedene Variablentypen. Eure Aufgabe ist es nun, zu den verschiedenen Funktionen die passenden Parameter aus dem Variablen herzustellen. Falls eine Variable nicht ohne großartige Konversationen verwendet werden kann tragt ihr bitte ein ✗ ein.

Als Beispiel dient hierfür die erste Zeile.

	int i	int *j	int const * const k	int **l	const int *m
op1(int *)	x	j	✗	*l	✗
op2(int)					
op3(int &)					
op4(const int **)					

Tabelle 1: Tabelle für Übergabewerte Aufgabe

---

## Übungen für den 2. Tag

---

### Aufgabe 2 Arrays und Zeigerarithmetik

---

Arrays sind zusammenhängende Speicherbereiche, die mehrere Variablen von gleichem Typ speichern können. Arrays werden in C++ folgendermaßen angelegt: `<Typ> <name>[<Größe>];`, zum Beispiel:

```
int arr[10];    // array of 10 integers
```

Falls das Array global ist, muss die Größe eine konstante Zahl sein, falls das Array in einer Funktion auf dem Stack angelegt wurde, kann die Größe auch durch eine Variable vorgegeben werden. Auf jeden Fall bleibt diese während der Existenz des Arrays konstant und kann sich nach dem Anlegen nicht mehr ändern.

Ein Array kann direkt bei der Deklaration initialisiert werden:

```
int arr[5] = {1, 2, 3, 4, 5};
```

Man kann die Größe optional auch weglassen, in diesem Fall wird sie der Compiler anhand der angegebenen Elemente selbst ermitteln.

Man kann auf die einzelnen Elemente des Arrays wie gewohnt über `arr[i]` zugreifen.

Arrays und Zeiger sind in C++ stark miteinander verwandt. So ist der **Bezeichner** des Arrays gleichzeitig die **Adresse des ersten Elements**. Somit kann man sowohl durch `*arr` als auch durch `arr[0]` auf das erste Element zugreifen. Analog dazu kann man auch einen Zeiger auf das erste Element anlegen:

```
int *pArr = arr;
```

Da die Elemente eines Array direkt hintereinander stehen, kann man den Zeiger inkrementieren, um zum nächsten Element zu gelangen (sogenannte Pointerarithmetik). Beispiel:

```
int *pArr = arr;
std::cout << "Address of first element: " << pArr << std::endl;
std::cout << "Address of second element: " << pArr+1 << std::endl;
std::cout << "Address of third element: " << pArr+2 << std::endl;
```

Somit kann man auf beliebige Elemente des Array über den Zeiger zugreifen:

```
*(pArr + 0);    // first element
*(pArr + 1);    // second element
*(pArr + 2);    // third element
++pArr;         // increment pointer by 1
*(pArr + 0);    // second(!) element of arr
*(pArr + 2);    // fourth(!) element of arr
```

Tatsächlich ist `*(p+i)` in **jeder Hinsicht äquivalent** zu `p[i]`. Das bedeutet, dass man sowohl auf das *i*-te Element eines Arrays über `*(arr + i)` zugreifen kann als auch über `pointer[i]` auf das Element, auf welches der Zeiger `pointer+i` zeigt! In C++ findet keine automatische Bereichsprüfung bei Arrayzugriffen statt. Du bist als Programmierer selbst dafür verantwortlich, dass niemals auf ein Element außerhalb der Array-Grenze zugegriffen wird. Falls doch, kann es zu Programmabstürzen oder unerwünschten Effekten wie Buffer-Overflows kommen, die ein erhebliches Sicherheitsrisiko darstellen. Bevorzuge deshalb Container-Klassen wie `std::vector` (oder `std::array` ab C++11) aus der Standardbibliothek anstelle von „rohen“ Arrays. Beachte außerdem, dass der `delete[]`-Operator zwar das Array löscht, den Zeiger jedoch **nicht** auf NULL setzt. Dabei entsteht ein *Dangling Pointer*, welcher dazu führen kann, dass später im Programm auf Speicherstellen zugegriffen wird, die nicht reserviert sind. Setze deshalb Zeiger nach einem `delete/delete[]` sofort auf NULL, um Speicherfehler zu vermeiden.

Um die Größe eines Arrays zu ermitteln, kannst du den `sizeof()`-Operator benutzen. Dieser gibt generell die Anzahl der Bytes an, die eine Variable verbraucht. Da einzelne Array-Elemente größer als ein Byte sein können, muss die Gesamtgröße des Arrays durch die Größe eines Elements geteilt werden, um auf die Anzahl der Elemente zu kommen.

```
int arr[10];
std::cout << sizeof(arr) << std::endl; // 40 on a typical 32 or 64-bit machine
int len = sizeof(arr) / sizeof(arr[0]);
std::cout << len << std::endl; // always 10
```

Beachte, dass `sizeof()` **nicht** dazu verwendet werden kann, um die Größe des Arrays herauszufinden, auf die ein Zeiger zeigt. In diesem Fall wird `sizeof()` nämlich die **Größe des Zeigers** und nicht die Größe des Arrays liefern!

---

## Übungen für den 2. Tag

---

```
int arr[10];
int *pArr = arr;
std::cout << sizeof(pArr) << std::endl; // 4 on 32-bit machine, 8 on 64-bit
```

---

### Aufgabe 2.1 Arrays anlegen

---

Lege in der main-Funktion ein `int`-Array mit 10 Elementen an, und initialisiere es mit den Zahlen 1 bis 10. Iteriere in einer Schleife über das Array und gib alle Elemente nacheinander aus.

---

### Aufgabe 2.2 printElements implementieren

---

In C und C++ kann man Arrays nicht direkt an Funktionen übergeben. Stattdessen übergibt man einen Zeiger auf das erste Element des Arrays. Aufgrund der Äquivalenz von `*(p+i)` und `p[i]` kann man in der Funktion den Zeiger syntaktisch wie das Original-Array verwenden.

Schreibe eine Funktion, die einen `const`-Zeiger auf das erste Element eines Arrays bekommt und alle Elemente ausgibt. Da die Funktion nur anhand des Zeigers keine Möglichkeit hat zu wissen, wie groß das Array ist, muss die Größe des Arrays durch einen weiteren Parameter übergeben werden<sup>1</sup>:

```
void printElements(const int *const array, const unsigned int size);
```

---

### Aufgabe 2.3 Offset-basierte Ausgabe

---

Wie wir vorher gesehen haben, kann man mit Zeigern auch rechnen und diese nachträglich ändern. Anstatt mit einem Index das Array zu durchlaufen, kann man stattdessen bei jeder Iteration den Zeiger selbst inkrementieren!

```
for(const int *p = array; p != array + 10; ++p) {
    int i = *p;    // *p contains current element
    // ...
}
```

Schreibe die Funktion aus der vorherigen Aufgabe so um, dass sie einen laufenden Zeiger anstatt eines Indexes verwendet.

---

### Aufgabe 2.4 Iterator-basierte Ausgabe

---

Ebenso kann man auch die Arraygröße auf eine andere Weise übergeben, indem man die Adresse des Elements nach dem letzten Element angibt. Dadurch werden Schleifen der folgenden Form möglich.

```
for(const int *p = begin; p != end; ++p) {
    int i = *p;    // *p contains current element
    // ...
}
```

möglich. Schreibe die Funktion aus der vorherigen Aufgabe entsprechend um. Vergiss nicht, den Zeiger als `const` zu definieren, da Elemente nur gelesen werden. Du kannst hier `const` doppelt verwenden, um auch sicherzustellen, dass der end-Zeiger nicht verändert wird.

---

### Aufgabe 2.5 Subarrays ausgeben

---

Die obige Methode, über Elemente eines Arrays zu iterieren, mag dir zunächst etwas ungewöhnlich erscheinen. Sie hat jedoch den Vorteil, dass man anstatt des ganzen Arrays auch kleinere zusammenhängende Teile davon an Funktionen übergeben kann, indem man Zeiger auf die entsprechenden Anfangs- und Endelemente setzt. Beispiel:

```
int arr[10];
printElements(arr+5, arr+8); // Print elements with index 5, 6, 7
```

Experimentiere etwas mit dieser Übergabemethode in deiner eigenen Funktion!

---

<sup>1</sup> Statt `unsigned int` wird oft der Standard-Typ `size_t` genutzt (z.B. in `std::vector`).

---

## Übungen für den 2. Tag

---

---

### Aufgabe 2.6 Arrays auf dem Heap

---

Bisher haben wir das Array auf dem Stack angelegt. Mit `new[]` kann man ein Array auf dem Heap erzeugen. Dabei wird die Adresse des ersten Elements in einem Zeiger gespeichert. Mittels `delete[]` muss man den belegten Speicher nach Benutzung freigeben. Beispiel:

```
int *pArr = new int[10]; // size can be a variable
doSomethingWith(pArr, 10);
delete[] pArr; // <-- notice the [] !
```

Beachte die `[]` nach `delete`. Diese bewirken, dass das gesamte Array und nicht bloß das erste Element gelöscht wird. Ein Anwendungsfall von dynamischen Arrays sind Funktionen, die ein Array von vorher unbekannter Größe zurückgeben. Schreibe eine Funktion, die beliebig viele Zahlen von der Konsole mittels `std::cin` einliest. Der Benutzer soll dabei zuvor gefragt werden, wie viele Zahlen er eingeben möchte. Speichere die Zahlen in einem dynamisch angelegten Array ab und lasse die Funktion einen Zeiger darauf zurückgeben. Hier ist ein Beispiel wie `std::cin` zu verwenden ist:

```
unsigned int size;
std::cout << "Größe: ";
std::cin >> size;
std::cout << "Gewählte Größe: " << size << std::endl;
```

Zusätzlich zum Zeiger muss die Funktion auch die Möglichkeit haben, ihrem Aufrufer die Größe des angelegten Arrays mitzuteilen. Füge der Funktion deshalb einen weiteren Parameter hinzu, in dem entweder per Referenz oder per Zeiger eine Variable übergeben wird, um dort die Größe abzulegen<sup>2</sup>.

Gib die eingelesenen Werte auf der Konsole aus. Vergiss nicht, am Ende den Speicher freizugeben.

---

<sup>2</sup> Du merkst sicherlich schon jetzt, dass es umständlich/fehleranfällig ist, wenn man die Größe eines Arrays separat speichern und übergeben muss.

## Übungen für den 2. Tag

### Aufgabe 3 Verkettete Listen

In dieser Aufgabe wollen wir eine doppelt verkettete Liste von Integern implementieren. Dazu brauchen wir zwei Klassen: `List` stellt ein Element der Liste mit dessen Inhalt dar und `List` speichert die Zeiger auf Anfangs- und Endelemente und bildet den eigentlichen Zugangspunkt für die Liste.

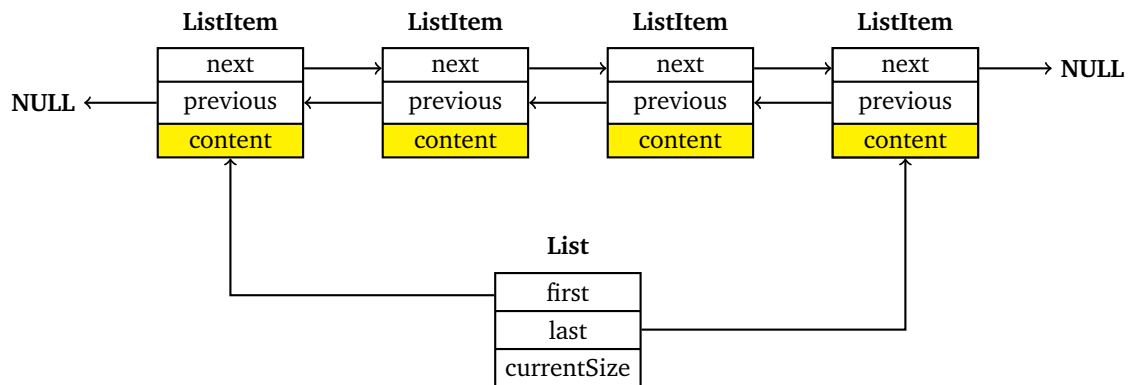


Abbildung 1: Linked List

Wir werden am Tag 4 auf dieser Aufgabe aufbauen und die Liste um weitere Funktionen erweitern. Behalte dies bitte im Hinterkopf und lösche deine Lösung nicht. Falls du mit dieser Aufgabe bis dahin nicht fertig sein solltest, kannst du natürlich auch die Musterlösung als Ausgangspunkt nehmen.

#### Aufgabe 3.1 Klasse ListItem

Implementiere die Klasse `ListItem`, welche die zu speichernde Zahl sowie Verweise auf das vorherige und nächste `ListItem` als Attribute hat. Verwende dazu Zeiger und keine Referenzen, da Referenzen nachträglich nicht mehr geändert werden können. Auch können Referenzen nicht NULL sein, was in unserem Fall nötig ist, um zu markieren, dass ein Element keine Vorgänger oder Nachfolger hat.

Der Konstruktor sollte sowohl seine eigenen `next` und `previous` Zeiger initialisieren, als auch die seiner Vorgänger- und Nachfolgerelemente. Die Methode `getContent()` soll eine Referenz auf den Inhalt zurückgeben, damit dieser durch eine Zuweisung modifiziert werden kann.

```
class ListItem {
public:
    /**
     * create a list item between two elements with a given content
     * (also modify previous->next and next->previous)
     */
    ListItem(ListItem *prev, ListItem *next, int content);
    /**
     * delete this list item (also change previous->next and next->previous
     * to not point to this item anymore)
     */
    ~ListItem();
    int & getContent(); // get a reference to the contained data
    ListItem * getNext(); // get the next list item or NULL
    ListItem * getPrevious(); // get the previous list item or NULL
private:
    ListItem *previous; // previous item in list
    ListItem *next; // next item in list
    int content; // content of this list item
};
```

---

## Übungen für den 2. Tag

---

---

### Aufgabe 3.2 Privater Copy-Konstruktor

---

Unsere `ListItem` Klasse hat einen kleinen Design-Fehler: Da wir keinen Copy-Konstruktor definiert haben, generiert der Compiler automatisch einen. Dieser kopiert einfach die einzelnen Attribute des Ursprungsobjekts (sogenannte „flache“ Kopie/Shallow Copy). In unserem Fall ergibt das Kopieren eines `ListItems` jedoch semantisch keinen Sinn, weil dabei ein hängendes `ListItem` entstehen würde, welches nicht mit der Liste verknüpft ist, aber dennoch auf andere Items der Liste zeigt.

Deklariere in der Headerdatei einen **private** Copy-Konstruktor und einen **private operator=**. Dadurch können beide nie aufgerufen werden und der Compiler kann dies zur Kompilierzeit überprüfen.

```
private:
    ListItem(const ListItem &other);    // private copy constructor (without implementation)
    ListItem& operator=(const ListItem &other); // private assignment operator (w/o implementation)
```

#### Hinweise

- Alternativ kann man ab C++11 Funktionen explizit löschen:  
`ListItem(const ListItem &other) = delete;`  
`ListItem& operator=(const ListItem &other) = delete;`

---

### Aufgabe 3.3 Klasse List

---

Implementiere nun die Klasse `List`. Achte bei den Methoden zum Einfügen und Entfernen von Elementen darauf, dass bei einer leeren Liste eventuell sowohl die `first` als auch `last` Zeiger modifiziert werden müssen. Vergiss nicht, `currentSize` bei jeder Operation entsprechend anzupassen. Falls die Liste leer ist, sollten `deleteFirst()` und `deleteLast()` einfach nichts ändern<sup>3</sup>.

#### **operator<<** implementieren

Implementiere außerdem den **operator<<**, um bequem Listen auf der Kommandozeile auszugeben. Die übergebene Referenz ist – entgegen der üblichen Konvention für **operator<<** – nicht **const**, da wir ansonsten entsprechend eine **const**-Version des `ListIterator` benötigen würden.

Vergiss hier nicht, **operator<<** als **friend** von `List` zu deklarieren (wie zuvor bei `Vector`).

```
class List {
public:
    List();                // create an empty list
    ~List();               // delete the list and all of its elements
    List(const List &other); // create a copy of another list
    void appendElement(int i); // append an element to the end of the list
    void prependElement(int i); // prepend an element to the beginning of the list
    void insertElementAt(int i, int pos); // insert an element i at position pos
    int getSize() const;    // get the number of elements in list
    int & getNthElement(int n); // get content of the n-th element.
    int & getFirst();        // get content of the first element
    int & getLast();         // get content of the last element
    int deleteFirst();       // delete first element and return it (return 0 if empty)
    int deleteLast();        // delete last element and return it (return 0 if empty)
    int deleteAt(int pos);   // delete element at position pos
private:
    ListItem *first, *last; // first and last item pointers (nullptr if list is empty)
    int currentSize;        // current size of the list
};

#include <iostream>
```

---

<sup>3</sup> Lieber würde man hier einen Fehler werfen, aber Exceptions haben wir an dieser Stelle noch nicht behandelt.



---

## Übungen für den 2. Tag

---

```
/** Print the given list to the stream. N.B. list should actually be const but then we would need
    const ListIterators */
std::ostream &operator<<(std::ostream &stream, List &list);
```

---

### Aufgabe 3.4 Liste testen

---

Teste deine Implementation. Füge der Liste Elemente von beiden Seiten hinzu und lösche auch wieder welche. Kopiere die Liste und gib die Elemente nacheinander aus.

---

### Aufgabe 3.5 ListIterator

---

Bisher haben wir über `getNthElement()` auf die Elemente der Liste zugegriffen. Diese Methode kann insbesondere bei langen Listen sehr langsam sein. Deshalb werden wir einen Iterator schreiben, über den man auf die Listenelemente sequentiell zugreifen kann. Der Iterator soll dabei einen Zeiger auf das aktuell betrachtete Element der Liste halten.

Um den Zugriff möglichst komfortabel zu gestalten, werden wir den Iterator als eine Art Zeiger implementieren, den man über `++` und `--` in der Liste verschieben kann. Um auf ein Element zuzugreifen, überladen wir den Dereferenzierungsoperator `operator*`. Somit können wir unsere Liste ähnlich zu `std::vector` verwenden:

```
for (ListIterator iter = list.begin(); iter != list.end(); iter++) {
    cout << *iter << endl;
}
```

### Konstruktor und Operatoren

Beginne mit einer Grundversion des Iterators. Erstelle einen Konstruktor, der die Attribute des Iterators (Zeiger auf aktuelles Element und Zeiger auf die Liste) entsprechend initialisiert. Implementiere Vergleichsoperator `operator!=` sowie den Dereferenzierungsoperator `operator*`. Der Dereferenzierungsoperator solle den Inhalt des aktuellen Items zurückgeben. Du brauchst nicht zu prüfen, ob `item` tatsächlich auf ein gültiges Element zeigt (Das machen/können Iteratoren aus der Standardbibliothek übrigens auch nicht!). Zum Vergleichen zweier Iteratoren prüfe, ob die `item` und `list` Zeiger identisch sind. Vergleiche nicht den Inhalt der Items, da der Vergleich auch dann funktionieren soll, wenn `item` NULL ist, wenn der Iterator also auf kein Element zeigt.

```
class ListIterator {
public:
    // create a new list iterator pointing to an item in a list
    ListIterator(List *list, ListItem *item);
    // get the content of the current element
    int& operator*();
    // check whether this iterator is not equal to another one
    bool operator!=(const ListIterator &other) const;
private:
    List *list;
    ListItem *item;
};
```

### Zugriff von außen: ListIterator als friend-Klasse

Du wirst in den folgenden Methoden auf private Attribute von `List` zugreifen müssen. Um dies zu ermöglichen, könnte man öffentliche Getter für die Items der Liste schreiben. Dadurch würde jedoch jeder die Möglichkeit bekommen, direkt auf die `ListItems` der Liste zuzugreifen, was dem Geheimnisprinzip zuwiderläuft. Deshalb werden wir `ListIterator` stattdessen explizit erlauben, auf `private`-Attribute der Liste zuzugreifen. Dazu müssen wir `ListIterator` als `friend` von `List` deklarieren. Füge dazu folgende Zeile (an beliebiger Stelle, üblich ist der Anfang der Klasse) zur Klassendefinition von `List` hinzu:

```
friend class ListIterator;
```

---

## Übungen für den 2. Tag

---

### Iterator vorwärts bewegen mittels `operator++`

Implementiere den `operator++` zum Inkrementieren des Iterators. Falls der Iterator zuvor auf kein Item zeigte (`item == NULL`), soll er nun auf das erste Element der Liste gesetzt werden. Die Prototypen dazu lauten:

```
ListIterator& operator++();           // increment this iterator and return itself (prefix ++)  
ListIterator operator++(int);        // increment this iterator and return the previous (postfix ++)
```

Bei der Überladung des `operator++` muss eine Sonderregelung beachtet werden. Dieser Operator kann sowohl als Postfix (z.B. `iter++`) als auch Präfix (z.B. `++iter`) verwendet werden. Um den Compiler darüber zu informieren, welche Variante wir überladen, wird beim Postfix-Operator ein Dummy-Parameter vom Typ `int` definiert. Dieser dient nur der syntaktischen Unterscheidung und hat keine weitere Bedeutung.

Beachte außerdem, dass bei Präfix-Operationen der Iterator sich selbst zurückgeben sollte, während der Postfix-Operationen eine Kopie des Iterators zurückgibt, die auf das vorherige Element zeigt. Das ist auch der Grund, warum die Präfix-Form von `operator++` (und `operator--`) effizienter ist als die Postfix-Form. Daher sollte die Präfix-Form dieser Operatoren die bevorzugte Variante sein, falls kein besonderer Grund für die Postfix-Form vorliegt.

Zum besseren Verständnis ist ein Teil der Implementation gegeben:

```
// Prefix ++ -> increment iterator and return it  
ListIterator& ListIterator::operator++() {  
    if (item == NULL) {  
        item = ... // set item to first item of list  
    }  
    else {  
        item = ... // set item to next item of current item  
    }  
    return *this; // return itself  
}  
  
// Postfix ++ -> return iterator to current item and increment this iterator  
ListIterator ListIterator::operator++(int) {  
    ListIterator iter(list, item); // Store current iterator  
    if (item == NULL) {  
        item = ... // set item to first item of list  
    }  
    else {  
        item = ... // set item to next item of current item  
    }  
    return iter; // return iterator to previous item  
}
```

### Iterator rückwärts bewegen mittels `operator--`

Überlade auf die gleiche Weise auch den `operator--` sowohl in Postfix als auch Präfix-Form.

### Iteratoren in List erzeugen

Nun ist unsere Implementation fast komplett und wir brauchen nur noch Methoden, um Iteratoren zu erzeugen. Implementiere dazu die folgenden Methoden innerhalb der `List` Klasse, um Iteratoren auf das erste und letzte Element der Liste zu erzeugen.

```
ListIterator begin();           // return an iterator pointing to the first element  
ListIterator end();             // return an iterator pointing to the element after the last one
```

Höchstwahrscheinlich wirst du Probleme bei der Kompilierung haben. Dies liegt an der zirkulären Abhängigkeit zwischen `List` und `ListIterator`. Gehe dazu folgendermaßen vor: Verschieben die `#include` Anweisungen für die Header von `List` und `ListItem` aus `ListIterator.h` nach `ListIterator.cpp` und füge in `ListIterator.h` folgendes hinzu

```
class ListItem;  
class List;
```

---

## Übungen für den 2. Tag

---

Dies sind Vorwärtsdeklarationen (**Forward Declaration**), die dem Compiler sagen, dass die Klassen existieren, aber später definiert werden. Nun kannst problemlos `ListIterator.h` in `List.h` einbinden.

---

### Aufgabe 3.6 Liste mit ListIterator testen

---

Teste deine Implementation. Erstelle eine Liste, füge Elemente hinzu und iteriere über Listenelemente:

```
for (ListIterator iter = list.begin(); iter != list.end(); iter++) {  
    cout << *iter << endl;  
}
```

Warum kann man **nicht** rückwärts durch die Liste iterieren, indem man einfach die Aufrufe `list.begin()` und `list.end()` tauscht und `iter--` statt `iter++` verwendet? Denke daran, worauf die von `begin()` und `end()` zurückgegebenen Iteratoren zeigen.

#### Hinweise

- In der Standardbibliothek gibt es hierfür `rbegin()` und `rend()`

---

## Übungen für den 2. Tag

---

### Aufgabe 4 Smart Pointers

---

In dieser Aufgabe werden wir uns mit der Benutzung von Smart Pointers vertraut machen. Dazu werden wir die Smart Pointer Klassen `std::shared_ptr` und `std::weak_ptr` verwenden. Binde hierfür den Systemheader `memory` ein.

---

#### Aufgabe 4.1

---

Erstellen eine Klasse `TreeNode`, die einen Knoten eines Binärbaums darstellt. Jeder Knoten hat einen Inhalt vom Typ `int` sowie einen Zeiger auf seine beiden Kindknoten. Statt „roher“ Zeiger verwenden wir Smart Pointers, die das Speicher-  
management übernehmen. Dadurch wird es nicht nötig sein, Kindknoten manuell zu löschen. Sie werden automatisch entfernt, sobald der Wurzelknoten gelöscht ist und keine Zeiger mehr auf den Kindknoten zeigen.

```
#include <memory>
class TreeNode;

typedef std::shared_ptr<TreeNode> TreeNodePtr; // typedef for better readability

class TreeNode {
public:
    /** create a new tree node and make it shared */
    static TreeNodePtr createNode(int content, TreeNodePtr left = TreeNodePtr(), TreeNodePtr right =
        TreeNodePtr());
    ~TreeNode();
private:
    TreeNode(int content, TreeNodePtr left, TreeNodePtr right); // create a tree node
    TreeNodePtr leftChild, rightChild;                        // left and right child
    int content;                                                // node content
};
```

Der Konstruktor von `TreeNode` privat, weil nur die Smart Pointer die Verantwortung für die Lebenszeit eines Objektes übernehmen sollen und bestimmen, wann es gelöscht wird. Würde man `TreeNode`-Objekte direkt auf dem Stack anlegen, kann es passieren, dass der Objektdestruktor mehrmals aufgerufen wird – einmal vom Smart Pointer und einmal beim Verlassen der Funktion. Ebenso sollten wir keine Rohzeiger auf das Objekt erzeugen, da diese das Speicher-  
management der Smart Pointer umgehen. Stattdessen stellen wir eine statische Methode bereit, um `TreeNode`-Objekte auf dem Heap zu erzeugen und diese direkt einem Smart Pointer zu übergeben.

Implementiere den Konstruktor, Destruktor sowie `createNode`. Der Konstruktor sollte die Attribute entsprechend initialisieren. Schreibe auch eine Textausgabe, die den Zeitpunkt der Erzeugung eines `TreeNode`s deutlich macht. Der Destruktor braucht die Kindknoten nicht zu löschen, da dies bei der Zerstörung des Elternknotens automatisch geschieht. Füge auch hier eine Textausgabe ein, die die Zerstörung des Objekts sichtbar macht.

Das Schlüsselwort **static** sowie die Default-Parameter müssen bei der Implementation der Methode ausgelassen werden. Der Smart Pointer für die Rückgabe wird mit einem Zeiger auf ein `TreeNode`-Objekt initialisiert. Somit lautet der Methodenrumpf

```
TreeNodePtr TreeNode::createNode(int content, TreeNodePtr left, TreeNodePtr right) {
    return TreeNodePtr(new TreeNode(...));
}
```

#### Hinweise

- Über `std::make_shared(<Konstruktorparameter>)` kann man auch einen Smart Pointer erhalten.
- Zur Dokumentation von `typedef` könnt ihr den Tag `@typedef` verwenden. Sonst verhält es sich mit dem Dokumentieren so wie immer.

---

#### Aufgabe 4.2

---

Teste, ob die einzelnen Knoten tatsächlich gelöscht werden, sobald kein Zeiger mehr auf den Elternknoten zeigt. Erstelle dafür einen kleinen Baum:

---

## Übungen für den 2. Tag

---

```
TreeNodePtr node = TreeNode::createNode(1, TreeNode::createNode(2), TreeNode::createNode(3));
```

Führe das Programm aus und beobachte die Ausgabe. Sobald `main` verlassen wird, wird der Zeiger `node` gelöscht, und somit auch das dahinterliegende `TreeNode`-Objekt mit all seinen Kindknoten.

Um ganz sicher zu gehen, dass der Baum tatsächlich beim Löschen des letzten Zeigers zerstört wurde und nicht etwa durch das Beenden des Programms, kannst du `node` mit einem anderen Baum überschreiben. Füge in diesem Fall am Ende des Programms eine Textausgabe hinzu, damit ersichtlich wird, dass der erste Baum noch vor Verlassen der `main` gelöscht wurde.

---

### Aufgabe 4.3

---

Nun wollen wir `TreeNode` so erweitern, dass jeder Knoten Kenntnisse über seinen Elternknoten besitzt. Füge das Attribut

```
TreeNodePtr parent;    // parent node
```

hinzu. Da der Elternknoten beim Erzeugen eines `TreeNode`s undefiniert ist, brauchst du den Konstruktor nicht zu ändern. `parent` wird dann automatisch mit `NULL` initialisiert.

Implementiere die folgende Methode, die einem Knoten seinen Elternknoten zuweist:

```
void setParent(const TreeNodePtr &p);    // set parent of this node
```

#### Hinweise

- `p` wird in diesem Fall nur deshalb als `const` Referenz übergeben, da es verhältnismäßig aufwändig ist, einen Smart Pointer zu kopieren. Beachte, dass im obigen Fall der Smart Pointer selbst `const` ist, und nicht das Objekt, worauf er zeigt.

Jetzt muss noch `createNode()` modifiziert werden, sodass `setParent()` auf den Kindknoten aufgerufen wird. Da ein Smart Pointer den `operator*` und den `operator->` überladen hat, lässt er sich syntaktisch wie ein normaler Zeiger benutzen. Um zu überprüfen, ob ein Smart Pointer auf ein Objekt zeigt, kann dieser implizit nach `bool` gecastet werden. Somit lautet die neue Implementation von `createNode()`:

```
TreeNodePtr TreeNode::createNode(int content, TreeNodePtr left, TreeNodePtr right) {
    TreeNodePtr node(new TreeNode(content, left, right));
    if (left) {
        left-> ... ; // set parent node
    }
    if (right) {
        right-> ... ; // set parent node
    }
    return node;
}
```

---

### Aufgabe 4.4

---

Teste deine Implementation. Du brauchst dazu in `main` nichts zu ändern.

Erschreckenderweise siehst du nun, dass überhaupt keine `TreeNode`-Objekte mehr gelöscht werden. Die Ursache dafür ist die zirkuläre Abhängigkeit zwischen Kind- und Elternknoten. Denn selbst wenn sie keine Zeiger auf den Wurzelknoten eines Baumes haben, verweisen die Kindknoten noch immer darauf.

Um dieses Problem zu lösen, müssen die Verweise zum Elternknoten *schwach* (weak) sein. Ein Knoten darf gelöscht werden, wenn nur noch schwache Zeiger (oder keine) auf ihn verweisen. Verwende dazu `std::weak_ptr` und erstelle ein neues `typedef` für einen schwachen `TreeNode` Smart Pointer:

```
typedef std::weak_ptr<TreeNode> TreeNodeWeakPtr;
```

Ändere nun den Typ von `parent` auf `TreeNodeWeakPtr`. Es müssen keine weiteren Änderungen gemacht werden, da starke Zeiger (`shared_ptr`) implizit in schwache Zeiger (`weak_ptr`) umgewandelt werden können.

#### Hinweise

- **Faustregel:** Wenn ein Objekt ein anderes kontrolliert oder enthält, verwende Shared Pointer vom Container/Besitzer zum anderen Objekt und einen Weak Pointer für die umgekehrte Richtung.

---

## Übungen für den 2. Tag

---

### Aufgabe 4.5

---

Teste deine Implementation. Nun sollte sich `TreeNode` wie gewünscht verhalten.