

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



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Übungen für den 2. Tag

### Aufgabe 1 Zeiger und Referenzen Grundlagen

- a) Experimentiere mit Zeigern, Adressen und Referenzen. Mache dir eine Skizze, die verdeutlicht, wie Variablen und ihre Speicherabbilder zusammenhängen. Als Ausgangsbasis kann dieses 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;
```

- b) Seien *intVal* und *pIntVal* wie oben gegeben. Versuche die Bedeutung folgender Ausdrücke zu verstehen. Welche Regelmäßigkeiten stellst du fest?

Tipp: Gehe dabei von innen nach außen vor.

`*&intVal;`

`*&pIntVal;`

`&*pIntVal;`

`**&pIntVal;`

`*&*intVal;`

`&*pIntVal;`

`*&pIntVal;`

- c) Warum sind folgende Ausdrücke ungültig?

Tipp 1: Finde heraus, welchen Typ der Ausdruck hätte haben müssen.

Tipp 2: Nur tatsächlich angelegte Variablen haben Adressen. Ausdrücke wie `a + b` oder direkt kodierte Zahlenlitterale wie `42` haben keine Adresse.

`*intVal;`

`**pIntVal;`

`**&*pIntVal;`

`&*intVal;`

---

## Übung zum C/C++-Praktikum - Tag 2

---

&42;

- d) Schreibe eine Funktion *swap*, die zwei *int*-Variablen miteinander vertauscht. Probiere dabei beide möglichen Übergabevarianten (Call-by-Reference, Call-by-Pointer) aus.
- e) Sieh dir folgendes Programm an:

```
void foo(int& i) {
    int i2 = i;
    int &i3 = i;

    cout << "i_=" << i << endl;
    cout << "i2_=" << i2 << endl;
    cout << "i3_=" << i3 << endl;
    cout << "&i_=" << &i << endl;
    cout << "&i2_=" << &i2 << endl;
    cout << "&i3_=" << &i3 << endl;
}

int main() {
    int var = 42;
    cout << "&var_=" << &var << endl;
    foo(var);
}
```

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

---

### 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>];
```

z.B.

```
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:

## Übung zum C/C++-Praktikum - Tag 2

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

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

```
int *pArr = arr;
cout << "Address_of_first_element:_" << pArr << endl;
cout << "Address_of_second_element:_" << pArr+1 << endl;
cout << "Address_of_third_element:_" << pArr+2 << 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 + i); // ith 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!

Nun bist du an der Reihe!

- 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.
- 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);
```

- 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++) {
    // *p contains current element
    // ...
}
```

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

- 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 Form

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

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

## Übung zum C/C++-Praktikum - Tag 2

```
// ...  
}
```

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.

- e) 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 zusammenhängenden Teil 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!

- f) 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 *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 *cin* zu verwenden ist:

```
// #include<iostream>  
// using namespace std;  
  
unsigned int size;  
cout << "Größe: ";  
cin >> size;  
cout << "Gewählte Größe: " << size << 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.

Noch ein paar Worte zum Schluss dieser Aufgabe: 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` aus der STL anstelle von „rohen“ Arrays. Beachte außerdem, dass der Operator **delete[]** 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.

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

## Übung zum C/C++-Praktikum - Tag 2

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.

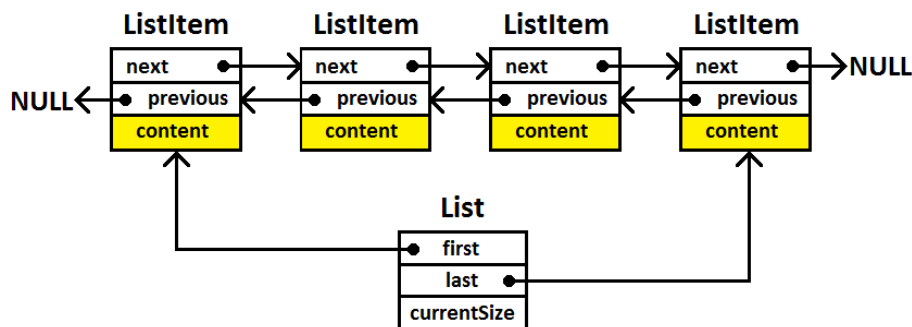
```
int arr[10];
cout << sizeof(arr) << endl; // 40 on 32-bit machine, 80 on 64-bit
int len = sizeof(arr) / sizeof(arr[0]);
cout << len << endl; // always 10
```

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!

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

### Aufgabe 3 Verkettete Listen

In dieser Aufgabe wollen wir eine doppelt verkettete Liste von Integern implementieren. Dazu brauchen wir zwei Klassen: *ListItem* 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.



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.

- a) **Klasse *ListItem*** Beginne mit der Klasse *ListItem*. *ListItem* soll die zu speichernde Zahl sowie Verweise auf das vorherige und nächste *ListItem* als Attribute haben. 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:
    /**
     * Creates a list item between two elements and stores the given content in the new item
     * Also modifies the corresponding previous and next pointers of prev and next item
     */
    ListItem(ListItem* prev, ListItem* next, int content);
```

---

## Übung zum C/C++-Praktikum - Tag 2

---

```
/**
 * Delete a list item. Changes also the pointers of previous and next elements to not
 * point to this item anymore.
 */
~ListItem();

/** Returns a reference to the contained data*/
int& getContent();

/** Returns the next list item, NULL if there is no next item*/
ListItem* getNext();

/** Returns the previous list item, NULL if there is no previous item*/
ListItem* getPrevious();

private:
/** Previous and next items in list */
ListItem *previous;
ListItem *next;

/** Content of the list item */
int content;
};
```

- b) **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 (Shallow Copy). In unserem Fall ergibt das Kopieren eines Items 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.

Deshalb werden wir das Kopieren von *ListItem*-Objekten verbieten, indem wir einen privaten Copy-Konstruktor deklarieren, ohne ihn zu implementieren. Dadurch kann der Copy-Konstruktor nie aufgerufen werden und der Compiler kann dies zur Compilezeit überprüfen.

Implementiere einen leeren privaten Copy-Konstruktor

```
private:
/** Private copy constructor (without implementation)*/
ListItem(const ListItem& other);
```

- c) **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>.

```
class List {
public:
/** Creates an empty list */
List();

/** Deletes the list and all its elements */
~List();
```

---

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

---

## Übung zum C/C++-Praktikum - Tag 2

---

```
/** Creates a copy of another list */
List(const List& other);

/** Appends an element to the end of the list */
void appendElement(int i);

/** Prepends an element to the beginning of the list. */
void prependElement(int i);

/** Inserts an element i at position pos.*/
void insertElementAt(int i, int pos);

/** Returns the number of elements in list */
int getSize() const;

/** Returns the n-th element. */
int& getNthElement(int n);

/** Returns the first element */
int& getFirst();

/** Returns the last element */
int& getLast();

/** Deletes first element and returns it. If list is empty, returns 0. */
int deleteFirst();

/** Delete last element and returns it. If list is empty, returns 0. */
int deleteLast();

/** Deletes element at position pos.*/
int deleteAt(int pos);

private:
/** First and last item pointers. NULL if and only if list is empty */
ListItem* first, *last;

/** Current size of the list */
int currentSize;
};
```

- d) Teste deine Implementierung. Füge der Liste Elemente von beiden Seiten hinzu und lösche auch wieder welche. Kopiere die Liste und gib die Elemente nacheinander aus.
- e) 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 sequenziell 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 \*. Somit können wir unsere Liste ähnlich zu *std::vector* verwenden:

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

---

## Übung zum C/C++-Praktikum - Tag 2

---

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 den Vergleichsoperator `!=` sowie den Dereferenzierungsoperator `*`. 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 STL-Iteratoren übrigens auch nicht!). Zum Vergleichen zweier Iteratoren prüfen Sie, 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, der Iterator also auf kein Element zeigt.

```
class ListIterator {
public:
    /** Creates a new list iterator pointing to an item in a list */
    ListIterator(List* list, ListItem* item);

    /** Returns element content of the current element. */
    int& operator*();

    /** Checks whether this iterator is not equal to another one */
    bool operator!=(const ListIterator& other) const;

private:
    List* list;
    ListItem* item;
};
```

Implementiere nun 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:

```
/** Increments this iterator and returns itself (prefix ++) */
ListIterator& operator++();

/** Increments this iterator but returns previous iterator (copy) (postfix ++) */
ListIterator operator++(int);
```

Bei der Überladung des `++` Operators muss eine Sonderregel 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.

Zum besseren Verständnis ist ein Teil der Implementierung 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
```



---

## Übung zum C/C++-Praktikum - Tag 2

---

```
    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
}
```

Du wirst in den Methoden auf private Attribute der Liste 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 Items 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. Fügen dazu folgende Zeile (an beliebiger Stelle, üblich ist der Anfang der Klasse) zur Klassendefinition von *List* hinzu:

```
friend class ListIterator;
```

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

Nun ist unsere Implementierung 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.

```
/** Returns an iterator pointing to the first element */
ListIterator begin();

/** Returns an iterator pointing to element after the last one*/
ListIterator end();
```

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;
```

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

- f) Teste deine Implementierung. 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.

---

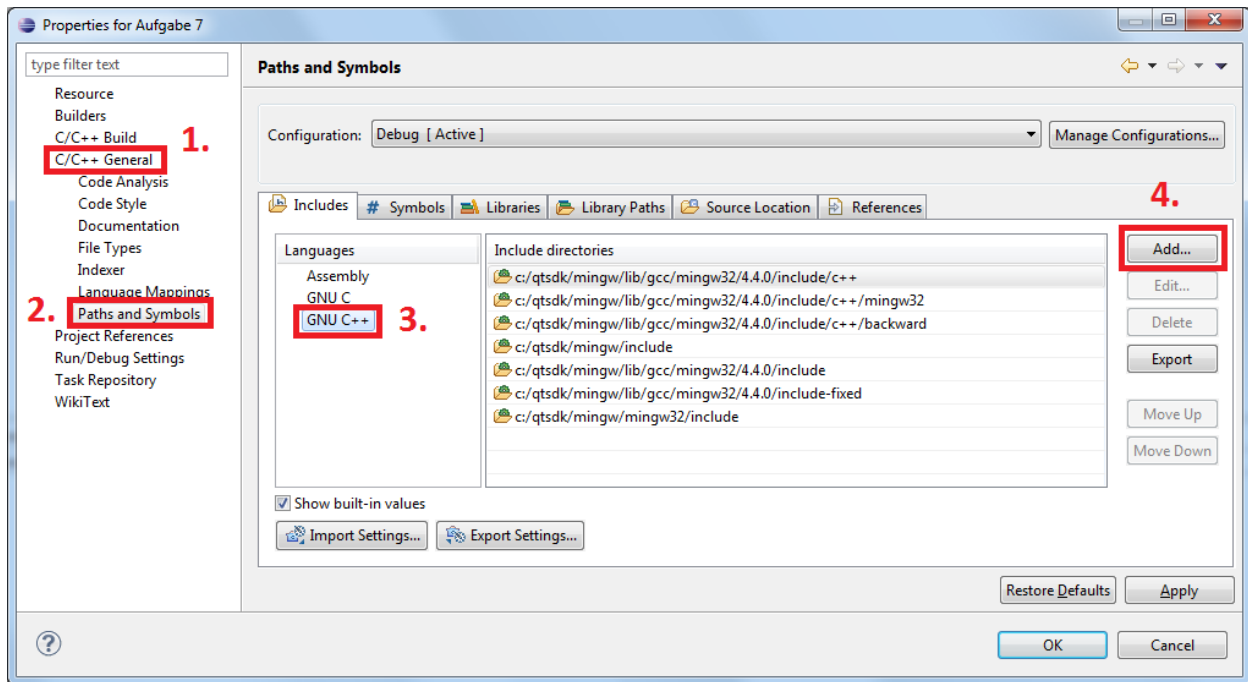
### Aufgabe 4 Smart Pointers

---

In dieser Aufgabe werden wir uns mit der Benutzung von Smart Pointers vertraut machen. Dazu werden wir die Smart Pointer Klassen *boost::shared\_ptr* und *boost::weak\_ptr* der Boost-Bibliothek verwenden. Boost wurde bereits mit dem zu diesem Praktikum dazugehörigen Eclipse-Installer installiert.

## Übung zum C/C++-Praktikum - Tag 2

- a) Erstelle ein neues Projekt und binde die Header-Ordner der Boost-Bibliothek zu dem Include-Pfad des Compilers ein. Klicke dafür mit der rechten Maustaste auf das Projekt und wähle **Properties** im Kontextmenü (Shortcut: **Alt+Enter**). Wähle dort **C/C++ General** → **Path and Symbols**. Markiere **GNU C++** und klicke auf **Add...**, um den Dialog zu öffnen. Der Pfad des Boost-Ordners ist bereits in der Variable **BOOST\_ROOT** gespeichert. Wähle im Dialog die Schaltfläche **Variables...** und klicke doppelt auf **BOOST\_ROOT**, sodass im Feld **Directory** der Eintrag **\${BOOST\_ROOT}** erscheint. Markiere das Kästchen **Add to all configurations**. Bestätige mit **OK** und schließe den Dialog



- b) 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 Speichermanagement 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 <boost/shared_ptr.hpp>

class TreeNode;

// typedef for better reading
typedef boost::shared_ptr<TreeNode> TreeNodePtr;

class TreeNode {
public:
    /** Creates a new tree node and makes it shared */
    static TreeNodePtr createNode(int content, TreeNodePtr left = TreeNodePtr(), TreeNodePtr
        right = TreeNodePtr());

    /** Tree node destructor */
    ~TreeNode();

private:
    /** Creates a tree node */
    TreeNode(int content, TreeNodePtr left, TreeNodePtr right);
};
```

---

## Übung zum C/C++-Praktikum - Tag 2

---

```
/** Left and right child */
TreeNodePtr leftChild, rightChild;

/** node content */
int 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 Speichermanagement 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 *TreeNodes* 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 Implementierung 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(...));
}
```

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

```
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 es ersichtlich wird, dass der erste Baum noch vor Verlassen der *main* gelöscht wurde.

- d) Nun wollen wir *TreeNode* so erweitern, dass jeder Knoten Kenntnisse über seinen Elternknoten besitzt. Füge das Attribut

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

hinzu. Da der Elternknoten beim Erzeugen eines *TreeNodes* 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:

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

*Hinweis:* *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

---

## Übung zum C/C++-Praktikum - Tag 2

---

Objekt, worauf er zeigt.

Jetzt muss noch `createNode()` modifiziert werden, sodass `setParent()` auf den Kindknoten aufgerufen wird. Da ein Smart Pointer die Operatoren `*` und `->` 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 Implementierung 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;
}
```

- e) Teste deine Implementierung. 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. Binde dazu den Header `boost/weak_ptr.hpp` ein und erstelle ein neues `typedef` für einen schwachen `TreeNode` Smart Pointer:

```
typedef boost::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.

- f) Teste deine Implementierung. Nun sollte sich `TreeNode` wie gewünscht verhalten.

---

### Aufgabe 5 Fortsetzung Aufzug

---

In dieser Aufgabe erweitern und verbessern wir unseren Aufzugsimulator, sodass das Kopieren von Personen wegfällt. Dies werden wir erreichen, indem wir nicht direkt mit `Person`-Objekten oder -Rohzeigern sondern mit Smart Pointern arbeiten. Dadurch müssen wir beim Verschieben von Personen in den Aufzug nur die Smart Pointer kopieren, während die `Person`-Objekte selbst bestehen bleiben.

Ein weiterer Vorteil ist, dass wir von jeder Person genau ein Exemplar im Speicher halten. Möchten wir beispielsweise den Namen einer Person ändern, ist dies überall, wo die Person auftaucht sofort und konsistent sichtbar. Nutzt man überall Kopien von Personen, haben wir keine Kontrolle darüber und wären gezwungen die Klasse `Person` immutabel zu machen.

*Hinweis:* In der nächsten Aufgabe hast du die Möglichkeit, die Performanz der alten und der neuen Implementierung zu vergleichen. Dazu ist es nötig, dass du jetzt eine Kopie von deinem aktuellen Code machst. In Eclipse kannst du ganze Projekte per Copy & Paste kopieren.

- a) **Refactoring mit Referenzen und `const`** Als Erstes verbessern wir die Sauberkeit des vorhandenen Codes mithilfe der bisher kennengelernten Mittel wie Referenzen und `const`. Es ist sinnvoll, dass du die Änderungen stückweise im Code durchführst und zwischendurch testest, ob alles noch korrekt funktioniert.

Deklariere dafür sämtliche Getter in `Building`, `Elevator`, `Floor` und `Person` als `const`, z.B. `Building::getFloor()` und `Elevator::getEnergyConsumed()`. Passe außerdem die Methode `Elevator::addPeople()` so an, dass die Liste `people` nicht mehr als Wert sondern als `const` Referenz übergeben wird.

---

## Übung zum C/C++-Praktikum - Tag 2

---

*Hinweis:* um über eine *const* Liste zu iterieren, verwende *vector<T>::const\_iterator* anstatt *vector<T>::iterator* als Iterator-Typ.

- b) Binde in deinem Projekt den Boost-Ordner in den Include-Pfad des Compilers ein (siehe vorige Aufgabe).
- c) Um nicht immer wieder *boost::shared\_ptr<Person>* schreiben zu müssen, definiere ein **typedef *PersonPtr*** für diesen Typen. Binde den Header *boost/shared\_ptr.hpp* in *Person.h* ein und definiere den neuen Typen *PersonPtr* hinter der Klassendefinition von *Person*:

```
typedef boost::shared_ptr<Person> PersonPtr;
```

- d) **Effizientere Listen** Ändere in der Klasse *Elevator* alle Vorkommen von *vector* nach *list* um, da wir nun eine Verkettete Liste verwenden werden, um Personen zu speichern. Dadurch kann man Personen auch in der Mitte der Liste effizient löschen.

Die *list*-Klasse enthält keine Methode *at()*. Diese ist auch gar nicht nötig: Wir traversieren die Liste stattdessen mit einem Iterator. Lösche dazu die Methode *getPerson()* und füge die folgende Methode hinzu, die eine *const* Referenz auf die enthaltenen Personen zurückgibt:

```
/** Returns a const reference to list of contained people */  
const std::list<PersonPtr>& getContainedPeople() const;
```

Dadurch kann von außen lesend auf die Leute im Aufzug zugegriffen werden. Ändere außerdem den Inhaltstyp des Containers von *Person* auf *PersonPtr*, da wir Smart Pointer auf Personen speichern werden und nicht die Personen direkt. Passen die Signaturen aller Methoden in *Elevator* entsprechend an.

- e) Jetzt müssen wir die Methode *removeArrivedPeople()* anpassen. Da wir beliebige Elemente aus *containedPeople* löschen können, brauchen wir den Umweg über die temporäre Liste *stay* nicht mehr.

Gehe dazu folgendermaßen vor: Iteriere mit einem Listeniterator vom Typ ***std::list<PersonPtr>::iterator*** über die Personen im Aufzug und prüfe für jede, ob sie an ihrem Zielstockwerk angekommen ist. Du kannst das Element, auf den der Iterator zeigt, durch den Dereferenzierungsoperator (*\*iter*) holen. Dieses Element ist selbst ein Smart Pointer. Deshalb muss der Iterator für den Zugriff auf die Person *doppelt* dereferenziert werden. Wenn die Person in ihrem Zielstockwerk angekommen ist, wird sie aus *containedPeople* gelöscht zu *arrived* hinzugefügt. Um ein Element zu löschen, verwende *containedPeople.erase(iter)*.

**Wichtig:** Der bisherige Iterator ist nach dem Löschen nicht mehr gültig. Die Methode *erase* gibt einen Iterator auf das Element hinter dem gelöschten zurück.

Als Grundgerüst kann folgendes Codeschnipsel dienen:

```
// create iterator for containedPeople  
... iter = containedPeople. ...;  
  
// iterate through all elements  
while (iter != ...) {  
    PersonPtr person = ... iter; // get person smart pointer at current position  
  
    // check whether person has reached it's destination Floor  
    if (...) {  
        // erase person pointer from containedPeople  
        // no need for ++iter since iter will already point to next item  
        ... = containedPeople.erase(iter);  
  
        // remember arrived person  
        ...  
    } else  
        ++iter; // check next person
```

---

## Übung zum C/C++-Praktikum - Tag 2

---

}

- f) Passe auch die Klassen *Floor* und *Building* entsprechend an, sodass nur noch Listen und Smart Pointer auf Personen verwendet werden.
- g) Passe die Simulation des Aufzugs entsprechend an. Du wirst auf die erste Person im Aufzug nun auf eine andere Art und Weise zugreifen müssen als vorher. Benutze die Methode *getContainedPeople()* des Aufzugs, um an die Liste der Personen zu kommen. Nun kannst du auf den Inhalt des ersten Elements mittels *front()* zugreifen. Vergiss nicht, dass dieser Inhalt ein *PersonPtr* und nicht die Person direkt ist. Entweder du dereferenzierst das Element doppelt und verwendest den *.*-Operator oder du nutzt wie üblich bei Pointern den *—>*-Operator.

Schau dir die Ausgabe an. Nun werden Personen nicht mehr kopiert, sondern nur noch gelöscht, sobald sie tatsächlich den Aufzug verlassen haben und kein Zeiger mehr auf sie zeigt.

---

### Aufgabe 6 Vergleich der alten und neuen Implementierung (optional)

---

Es ist natürlich interessant zu erfahren, ob sich der ganze Aufwand des Refactorings gelohnt hat.

#### Laufzeit

Eine relativ simple Art der Laufzeitmessung ist es, die verstrichene Prozessorzeit zu messen. Der Header *ctime* stellt hierfür die Funktion *clock()* zur Verfügung, die einen Zähler vom Typ *clock\_t* zurückgibt. Mithilfe der Konstanten *CLOCKS\_PER\_SEC* kann man aus der Anzahl von Prozessorzyklen die Laufzeit berechnen.

Erzeuge nun ein hinreichend großes Beispiel und teste dessen Laufzeit für die alte und neue Implementierung.

Es gibt auch ausgefeiltere Möglichkeiten, die Laufzeit zu messen. Dazu stellt Boost unter anderem den Header *boost/chrono.hpp* zur Verfügung. Für nähere Informationen siehe [http://www.boost.org/doc/libs/1\\_48\\_0/doc/html/chrono/users\\_guide.html](http://www.boost.org/doc/libs/1_48_0/doc/html/chrono/users_guide.html).

*Hinweis:* Die benötigten Boost-Bibliotheken sind auf den Pool-PCs nicht installiert.

#### Speicherverbrauch

Ein weiteres Argument gegen das Kopieren von Objekten kann der Speicherverbrauch sein. Das ist in unserem Fall allerdings weniger interessant, da die meisten kopierten Objekte relativ kurz leben und dann wieder gelöscht werden.

Im Gegensatz zur Laufzeit gibt es leider keinen sehr einfachen Weg, den Speicherverbrauch des Programms direkt auszugeben.

Du könntest hierfür kurz vor dem Ende von *main* die Ausführung pausieren (mittels *std::cin*) und dir im Task Manager ansehen, wie hoch der Speicherverbrauch des Programms ist – das ist aber sicherlich nur eine Notlösung.