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

TECHNISCHE UNIVERSITÄT DARMSTADT

## Zusatzaufgaben

### Aufgabe 1 Aufzugsimulator

In dieser Aufgabe soll ein Grundgerüst für den in der Vorlesung vorgestellten Aufzugsimulator geschaffen werden. Bei der Bearbeitung dieser Aufgaben geben wir die keinerlei zeitliche Vorgabe. Du kannst diese Aufgaben direkt an dem Tag lösen, an dem ihr die dafür benötigten Konzepte gelernt habt oder auch nach hinten verschieben und als ausführliche Klausurvorbereitung nutzen.

### Aufgabe 1.1 Klasse Person

Implementiere die Klasse `Person`, die eine Person mit einem gewünschten Zielstockwerk darstellt. Füge allen Konstruktoren und Destrukoren eine Ausgabe auf die Konsole hinzu, um später den Lebenszyklus der Objekte besser nachvollziehen zu können.

```
class Person {
public:
   Person(int destinationFloor);    // create a person with given destination
   Person(const Person& other);     // copy constructor
   ~Person();                       // destructor
   int getDestinationFloor() const; // get the destination floor of this person
private:
   int destinationFloor;            // destination floor of this person
};
```

### Aufgabe 1.2 Klasse Elevator

Implementiere die Klasse `Elevator`, die einen Aufzug mit einer beliebigen Anzahl an Personen darstellt.
Wenn sich der Aufzug bewegt, solltest du die verbrauchte Energie bei einer Bewegung sinnvoll anpassen. Addiere beispielsweise den Betrag der Differenz zwischen dem aktuellen und dem Zielstockwerk hinzu.

```
class Elevator {
public:
   Elevator(); // create an elevator at floor 0, no people inside and 0 energy consumed
   int getFloor();               // get number of floor the elevator is currently at
   double getEnergyConsumed();   // get consumed energy
   void moveToFloor(int floor);  // move the elevator to given floor (consumes energy)
   int getNumPeople();           // get number of people in Elevator
   Person getPerson(int i);      // get i-th person in Elevator
   void addPeople(std::vector<Person> people);  // add people to Elevator
   std::vector<Person> removeArrivedPeople();   // remove people which arrived
private:
   int currentFloor;                      // current floor number
   std::vector<Person> containedPeople;   // people currently in elevator
   double energyConsumed;                 // energy consumed
};
```

Um die Klasse `std::vector` aus der Standardbibliothek zu nutzen musst du noch den Systemheader `vector` einbinden. Der Container `std::vector` kapselt ein Array und stellt eine ähnliche Funktionalität wie Javas `Vector` Klasse bereit.

## Zusatzaufgaben

Der Typ in spitzen Klammern (<Person> in std::vector<Person>) ist ein Template-Parameter und besagt, dass in dem Container Person-Objekte gespeichert werden sollen.
Folgende Funktion der Klasse std::vector könnten dir von Nutzen sein. Weitere findest du z.B. unter http://www.cplusplus.com/reference/vector/vector/. Der Typ size_type ist der größtmöglichhe vorzeichenloser Integer, die die verwendete Plattform unterstützt.

```
size_type size() const;                // get size of the vector
reference at(size_type n);             // get the i-th element of the vector
void push_back(const value_type& val); // add an element to the vector
void clear();                          // remove all elements from the vector
```

**Hinweise**

- Da containedPeople leer initialisiert werden soll, brauchst du dafür keinen expliziten Aufruf in der Initialisierung.
- Um die Leute aussteigen zu lassen, die an ihrem Zielstockwerk angekommen sind, erstelle in der Methode zwei temporäre std::vector-Container stay und arrived. Iteriere nun über alle Leute im Aufzug und prüfe, ob das Zielstockwerk der Person mit dem aktuellen Stockwerk des Aufzugs übereinstimmt. Wenn ja, lasse die Person aussteigen, indem du sie zu der arrived-Liste mittels push_back() hinzufügst. Andernfalls muss die Person im Aufzug verbleiben (stay-Liste). Gib am Ende die arrived-Liste zurück, und ersetze containedPeople durch stay.

### Aufgabe 1.3 Klasse Floor

Implementiere die Klasse Floor, die ein Stockwerk mit einer beliebigen Anzahl an wartenden Personen darstellt.

```
class Floor {
public:
   int getNumPeople();                    // get the number of people on this floor
   Person getPerson(int i);               // get the i-th person on this floor
   void addWaitingPerson(Person h);       // add a person to this floor
   std::vector<Person> removeAllPeople(); // remove all persons from this floor
private:
   std::vector<Person> containedPeople;   // persons on this floor
};
```

### Aufgabe 1.4 Klasse Building

Schreibe eine Klasse Building, die einen Aufzug besitzt der sich zwischen einer definierbaren Menge an Stockwerken bewegt und Personen befördert.

```
class Building {
public:
   Building(int numberOfFloors); // create a Building with given number of floors
   int getNumOfFloors();         // get number of floors
   Floor& getFloor(int floor);   // get a certain floor
   Elevator& getElevator();      // get the elevator
private:
   std::vector<Floor> floors;    // floors of this building
   Elevator elevator;            // the elevator
};
```

### Aufgabe 1.5 Komfortfunktionen

Erweitere die Klasse Building um folgende public Funktionen, um die Benutzung des Simulators von außen zu vereinfachen und lange Aufrufketten wie

```
b.getElevator().addPeople(b.getFloor(b.getElevator().getFloor()).removeAllPeople());
```

zu vermeiden (*Law of Demeter*[1]). Der Simulator sollte nur mit Methoden der Klasse `Building` kommunizieren.

```
void letPeopleIn();                          // let people on current floor into elevator
void moveElevatorToFloor(int i);             // move the elevator to a given floor
void addWaitingPerson(int floor, Person p);  // add a person to a given floor
std::vector<Person> removeArrivedPeople();   // remove people which arrived at their destination
    from the elevator on the current floor
```

### Aufgabe 1.6 Beförderungsstrategie

Teste deine Implementation. Erstelle dazu zunächst ein Gebäude und füge einige Personen hinzu.

```
Building b(3);
b.addWaitingPerson(0, Person(2)); // person in floor 0 wants to floor 2
b.addWaitingPerson(1, Person(0)); // person in floor 1 wants to floor 0
b.addWaitingPerson(2, Person(0)); // person in floor 2 wants to floor 0
```

Implementiere nun folgende Beförderungsstrategie. Diese sehr einfache (und ineffiziente) Strategie fährt alle Stockwerke nacheinander ab, sammelt die Leute ein und befördert sie jeweils zu ihren Zielstockwerken.

**for** *Floor floor* ***in*** *Building* **do**
  Move `elevator` to Floor `floor`;
  Let all people on `floor` into `elevator`;
  **while** *elevator has people* **do**
    Move Elevator to destination Floor of first Person in Elevator;
    Remove arrived people;
  **end**
**end**

Gib am Ende auch die verbrauchte Energie aus. Schau dir die Ausgabe genau an und versuche nachzuvollziehen, warum Personen so oft kopiert werden. Denke daran, dass diese bei einer Übergabe als Argument kopiert werden.

#### Hinweise

- Falls du wie vorgeschlagen als Modell für den Energieverbrauch den Betrag der Differenz der abgefahrenen Stockwerke verwendest, solltest du am Ende `consumedEnergy = 8` erhalten.

[1] https://en.wikipedia.org/wiki/Law_of_Demeter

## Zusatzaufgaben

### Aufgabe 2 Fortsetzung Aufzugsimulator

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.

#### Hinweise

- Am Ende dieser Aufgabe hast du die Möglichkeit, die Performanz der alten und der neuen Implementation zu vergleichen. Dazu ist es nötig, dass du jetzt eine Kopie von deinem aktuellen Code machst. Gehe dabei folgendermaßen vor:
  - Lege eine Kopie deines Projektordners an.
  - Benenne dein Projekt um, z.B. in `Aufzug_Alt` (**Rechtsklick auf das Projekt → Rename Project**).
  - Importiere den kopierten Projektordner in CodeLite (**Workspace → Add an existing project**).
  - In deinem Workspace sollten jetzt zwei Projekte sein, `Aufzug_Alt` und `Aufzug`.
- Du wirst in den folgenden Aufgaben deinen Code stark verändern. Versuche, nicht alle Änderungen auf einmal zu machen, sondern gehe stückweise vor (also bspw. nur einen Getter umstellen und alle Folgefehler beheben).

### Aufgabe 2.1 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.
Es kann sein, dass du für bestimmte Zwecke weiterhin einen Getter brauchst, der dir ein nicht-`const` Objekt zurückgibt. Solche Getter sollten typischerweise `private` deklariert werden. Beispielsweise wird deine `Building`-Klasse zwei Getter für `Floor` enthalten:

```
class Building {
public:
    // ...

    /** Gets a certain, const floor */
    const Floor& getFloor(int floor) const;

private:
    /** Returns the floor with the given number.
     *  This non-const variant of the getter is for 'private' purposes only.
     */
    Floor& getFloor(int floor);

    // ...
};
```

#### Hinweise

- Um über eine `const` Liste zu iterieren, verwende `vector<T>::const_iterator` anstatt `vector<T>::iterator` als Iterator-Typ.

## Zusatzaufgaben

### Aufgabe 2.2

Um nicht immer wieder `std::shared\_ptr<Person>` schreiben zu müssen, definiere ein `typedef` `PersonPtr` für diesen Typen. Binde den Header `memory` in `Person.h` ein und definiere den neuen Typen `PersonPtr` hinter der Klassendefinition von `Person`:

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

### Aufgabe 2.3 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:

```
/** return a const reference to the 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. Passe die Signaturen aller Methoden in `Elevator` entsprechend an.

### Aufgabe 2.4

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 einen 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 auf das Element, auf den der Iterator zeigt, durch den Dereferenzierungsoperator (`*iter`) zugreifen. 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 und zu `arrived` hinzugefügt. Um ein Element zu löschen, verwende `containedPeople.erase(iter)`.

#### Hinweise

- 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:

```
... iter = containedPeople. ...;    // create iterator for 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
   }
}
```

## Zusatzaufgaben

### Aufgabe 2.5

Passe auch die Klassen `Floor` und `Building` entsprechend an, sodass nur noch Listen und Smart Pointer auf Personen verwendet werden.

### Aufgabe 2.6

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 2.7 Vergleich der alten und neuen Implementation (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 verstrichtene 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 Implementation.

**Hinweise**

- 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`.

**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.

**PJ:** Ich hätte gerne an irgendeiner Stelle den Aufruf dazu einmal `auto` zu verwenden.

## Zusatzaufgaben

### Aufgabe 3 Fortsetzung Aufzugsimulator

Unser bisheriger Aufzugsimulator hat eine feste Strategie, nach der die einzelnen Stockwerke abgefahren werden. Mithilfe von Polymorphie können wir den Simulator so erweitern, dass die Strategie austauschbar wird.

#### Aufgabe 3.1 Vorbereitung

Lagere die bereits existierende Simulation des Aufzugs aus der `main`-Funktion in eine eigene Funktion `runSimulation()` aus. Die Funktion sollte das volle Gebäude als Parameter entgegennehmen und eine Liste (`std::list<int>`) der angefahrenen Stockwerke zurückgeben. Überlege dir, auf welche Art das Gebäude idealerweise übergeben werden sollte. Teste deine Implementation.

#### Aufgabe 3.2 Klasse `ElevatorStrategy`

Implementiere die Klasse `ElevatorStrategy`. Diese soll die Basisklasse für verschiedene Aufzugstrategien sein. Damit die Strategie das Gebäude nicht selbst modifizieren kann, wird `Building` per `const` Pointer übergeben.

```
// Elevator strategy class: Determines to which floor the elevator should move next.
class ElevatorStrategy {
public:
   virtual ~ElevatorStrategy();
   virtual void createPlan(const Building*);    // create a plan for the simulation - the default
    implementation does nothing but saving the building pointer
   virtual int nextFloor() = 0;  // get the next floor to visit
protected:
   const Building *building;     // pointer to current building, set by createPlan()
};
```

#### Aufgabe 3.3 Eine einfache Aufzusstrategie

Implementiere eine einfache Aufzugstrategie, indem du eine neue Klasse erzeugst die von `ElevatorStrategy` erbt. Diese soll folgendermaßen vorgehen: Falls der Aufzug momentan leer ist, soll zum tiefsten Stockwerk gefahren werden, wo sich noch Personen befinden. Falls der Aufzug nicht leer ist, wird das Zielstockwerk einer der Personen im Aufzug ausgewählt.

#### Aufgabe 3.4 Implementation von `runSimulation`

Ändere nun `runSimulation()` entsprechend um, sodass die Simulation anhand der gegebenen Strategie durchgeführt wird. Folgender Pseudocode kann dir als Denkhilfe dienen:

**while** *People in Building or Elevator* **do**
  Calculate next floor;
  Move Elevator to next floor;
  Let all arrived people off;
  Let all people on floor into Elevator;
**end**

Teste die einfache Aufzugstrategie

#### Aufgabe 3.5 Neue Aufzugstrategien (optional)

Entwickle eine eigene Aufzugstrategien, indem du erneut eine neue Klasse erzeugst die von `ElevatorStrategy` erbt. Versuche, verschiedene Größen zu optimieren, wie z.B. die Anzahl der Stopps oder die verbrauchte Energie. Hierfür könnte Backtracking verwenden[2], eine einfache Methode, um eine optimale Lösungen durch Ausprobieren zu finden. Beachte, dass der Aufzug auch kopiert werden kann, um verschiedene Strategien zu testen.

[2] Siehe `http://de.wikipedia.org/wiki/Backtracking`

## Zusatzaufgaben

### Aufgabe 4 Makefiles

In dieser Übung erkunden wir, wie man Makefile-Projekte in CodeLite aufbaut. Wir bauen dafür einen Teil unseres Aufzugsimulators nach, um zu sehen, wo die Herausforderungen in echten Projekten mit Makefiles gelöst werden können.

#### Aufgabe 4.1 Projekt anlegen:

Wähle *Workspace* → *New project* und wähle als Projekttyp **CPPP/Empty Makefile**.
Das erzeugte Projekt enthält bereits eine Datei `Makefile` (im Ordner `resources`), diese hat aber noch keinen Inhalt. Make erwartet, dass das Makefile ein Target mit dem Namen `all` hat. Um unser Projekt zu testen, geben wir zunächst eine einfache Meldung auf der Kommandozeile aus. Lege nun das Target `all` an und füge den folgenden Befehl an (Vergiss dabei nicht den Tab vor jedem Befehl!):

```
all:
   @echo "Running all..."
```

Wenn du jetzt *Build* aufrufst, sollte in der Konsole in etwa Folgendes erscheinen:

```
----------Building project:[ mktest - Debug ]----------
making all...
====0 errors, 0 warnings====
```

#### Aufgabe 4.2 Erster Compilevorgang

Jetzt ist es an der Zeit, ein Programm mittels `make` zu kompilieren. Lege dazu eine C++-Sourcedatei `main.cpp` mit einer `main`-Funktion an, die etwas sinnvolles ausgibt.
Entgegen unserer bisherigen Erfahrung musst du nun manuell in `Makefile` eintragen, dass `main.cpp` gebaut werden soll. Ersetze die Dummy-Ausgabe daher durch einen Compiler-Aufruf an `g++`:

```
all:
   g++ -o main.exe main.cpp
```

Wenn du jetzt *Build* aufrufst, wird dein Programm kompiliert und als `main.exe` im Projekthauptverzeichnis abgelegt.

#### Aufgabe 4.3 Klasse `Building`:

Jetzt fügen wir die Klasse `Building` zu unserem Projekt hinzu, die allerdings nur minimale Funktionalität bietet:

```
#pragma once

#include <string>

class Building {
public:
    Building(unsigned int numFloors);
    const std::string toString() const;
private:
    unsigned int numFloors;
};
```

```
#include "Building.h"
#include <sstream>

Building::Building(unsigned int numFloors):
   numFloors(numFloors) {}

const std::string Building::toString() const{
   std::stringstream output;
   output << "A building with " << numFloors;
   output << " floors" << std::endl;
   return output.str();
}
```

Erzeuge in der `main`-Funktion eine zweistöckige Instanz von `Building` und gib diese mittels `Building::toString` auf der Konsole aus.
Damit das Projekt kompiliert, muss auch `Building` im Makefile eingetragen werden. Passe dazu den Compileraufruf an:

```
all:
    g++ -o main.exe main.cpp Building.cpp
```

Wenn du das Projekt gebaut hast und ausführst, sollte auf der Konsole eine Ausgabe deines Gebäudes erscheinen.

## Zusatzaufgaben

### Aufgabe 4.4 Compiler-Aufrufe auslagern:

In der Vorlesung haben wir gesehen, dass *make* anhand der Zeitstempel von Dateien dazu in der Lage ist, zu erkennen, wann ein Programmteil neu gebaut werden muss. Aktuell nutzen wir diese Möglichkeit noch nicht: Egal ob wir `main.cpp`, `Building.cpp` oder `Building.h` verändert haben, immer wird das gesamte Projekt neu gebaut. In diesem Schritt zerlegen wir die Abhängigkeiten zu den einzelnen Dateien.
Mache das Target `all` jetzt abhängig von den Objektdateien `main.o` und `Building.o` und erzeuge für jede Objektdatei ein eigenes Ziel, welches diese baut (Das Flag `-c` sorgt dafür, dass die Sourcedateien nur kompiliert, aber nicht gelinkt werden).

```
all: main.o Building.o
   g++ -o main.exe main.o Building.o

main.o: main.cpp
   g++ -c -o main.o main.cpp

Building.o: Building.cpp
   g++ -c -o Building.o Building.cpp
```

Baue das Projekt nun erneut, du solltest drei Aufrufe von `g++` sehen:

```
make all
g++ -c -o main.o main.cpp
g++ -c -o Building.o Building.cpp
g++ -o main.exe main.o Building.o
```

Baust du das Projekt nun erneut, so wird nur noch der Linker aufgerufen:

```
make all
g++ -o main.exe main.o Building.o
```

### Aufgabe 4.5 Linker-Aufruf auslagern

Wie können wir diesen an sich unnötigen Aufruf ebenfalls noch loswerden? Eine Lösung ist es, das Target `all` von `main.exe` abhängig zu machen und ein neues Ziel `main.exe` zu definieren:

```
all: main.exe

main.exe: main.o Building.o
   g++ -o main.exe main.o Building.o

# ...
```

Wenn du das Projekt jetzt baust, erhältst du erfreulicherweise die Rückmeldung, dass nichts zu tun ist:

```
make all
make: Nothing to be done for ‘all’.
```

### Aufgabe 4.6 Inkrementelles Bauen:

Wir erproben jetzt, wie sich Veränderungen an einer der drei Dateien auf die Ausführung von `make` auswirken. Mache nacheinander kleine Änderungen – das können auch Kommentare sein – an den Dateien `main.cpp`, `Building.h` und `Building.cpp` und baue das Projekt nach jeder Änderung.
Dir fällt auf, dass Änderungen an `Building.h` von `make` nicht bemerkt werden; die Datei taucht ja nirgendwo explizit im Makefile auf.
Wir sehen uns jetzt an, welche Tragweite dieses Problem haben kann.

## Zusatzaufgaben

### Aufgabe 4.7 Header als Abhängigkeiten

Du hast kennengelernt, dass man Implementationen auch `inline` in einem Header machen kann, zum Beispiel wenn diese klein sind.
Bewege `toString` nun nach `Buidling.h`:

```
#include <sstream>
// ...
const std::string toString() const{

    std::stringstream output;
    output << "A building with " << this->numFloors << " floors" << std::endl;
    return output.str();
}
```

Baue das Projekt; es kompiliert nicht! Warum? Genau aus dem Grund, dass `make` das Header-File nicht „kennt“. Jetzt gibt es im Projekt keine Definition von `toString` wie uns der Linker auch mitteilt:

```
main.o:main.cpp:(.text+0x5c): undefined reference to ‘Building::toString() const’
collect2: ld returned 1 exit status
```

Das Problem lässt sich lösen, indem wir im Makefile angeben, dass `main.o` nicht nur abhängig von `Building.cpp`, sondern auch von `Building.h` ist:

```
# ...
main.o: main.cpp Building.h
   g++ -c -o main.o main.cpp
# ...
```

Ist das eine schöne Lösung? Sicherlich nicht, denn ab sofort müssten wir manuell alle Header ins Makefile eintragen, die wir per `\#include` in eine Sourcedatei einbinden. Schlimmer noch: Wir müssten über rekursive Includes Bescheid wissen, z.B. wenn `Building.h` einen anderen veränderlichen Header wie `Floor.h` einbindet.
Glücklicherweise hilft uns g++ bei diesem Problem.

### Aufgabe 4.8 Header automatisch als Abhängigkeiten deklarieren:

Wir automatisieren jetzt die Erkennung von Headern als Abhängigkeiten. Lösche dazu die Abhängigkeit `Building.h` des Targets `main.o` und füge in den Compiler-Aufrufen die Parameter `-MMD -MP` hinzu. Binde außerdem die Dateien `Building.d` und `main.d` ein wie unten dargestellt:

```
all: main.exe

main.exe: main.o Building.o
   g++ -o main.exe main.o Building.o

main.o: main.cpp
   g++ -c -MMD -MP -o main.o main.cpp

Building.o: Building.cpp
   g++ -c -MMD -MP -o Building.o Building.cpp

-include Building.d main.d
```

Um den Effekt dieser Lösung zu sehen, müssen wir alle generierten Dateien löschen (`main.exe, Building.h, Building.cpp`). Das anschließende Bauen sollte nun funktionieren.
Der Trick ist, dass g++ beim Kompilieren für jede Sourcedatei ein Makefile generiert, das dessen eingebundene Header als Abhängigkeiten enthält (`main.d, Building.d`).
Wenn du jetzt Änderungen an der `toString`-Methode durchführst, werden diese anhand des Zeitstempels von `Building.h` erkannt.

## Zusatzaufgaben

### Aufgabe 4.9 Target *clean*

Bisher mussten wir hin und wieder die kompilierten Dateien manuell löschen, wenn wir unser Projekt neu bauen wollten. Diese Aufgabe lässt sich mittels `make` ebenfalls automatisieren.
Lege dazu ein neues Target `clean` ohne Abhängigkeiten an und füge einen entsprechenden Löschbefehl ein:

```
clean:
   rm -rf main.o Building.o main.d Building.d main.exe

.PHONY: clean
```

Das Spezial-Target `.PHONY` dient dazu, `make` zu signalisieren, dass `clean` keine Datei ist, die gebaut werden soll. Würden wir dieses Target auslassen und eine Datei mit Namen `clean` erzeugen, würde `make` die Regel nie ausführen, weil die Datei ja existiert und keine Abhängigkeiten besitzt. Du solltest `all` ebenfalls als `.PHONY` deklarieren. Probier' es ruhig aus!
Um `clean` ausführen zu können, öffne die **Make Target** View über **Window → Show View → Other... → Make/Make Target**.
Klicke rechts auf dein Projekt und wähle **New...**. Gib in das Feld **Make Target *clean*** ein und lasse den Rest unverändert. Unter deinem Projekt wurde nun ein neuer Knoten mit dem Namen `clean` eingefügt. Wenn du diesen doppelt anklickst, wird das Target `clean` ausgeführt.
Lege dir nun auch ein ausführbares Target für `all` an.

### Aufgabe 4.10 Generisches Compiler-Target

Dir ist sicherlich aufgefallen, dass wir zwei Targets haben, die mehr oder weniger identisch sind: `main.o` und `Building.o`. `make` bietet für solche Situationen generische Regeln an, die mittels Wildcards beschrieben werden.
Ersetze die beiden spezifischen Targets durch folgendes generisches:

```
%.o: %.cpp
   g++ -MMD -MP -c $< -o $@
```

Die etwas kryptischen Ausdrücke *$<* und *$@* werden durch die aktuelle Abhängigkeit und Target ersetzt.
Lösche alle automatisch generierten Dateien (`make clean`) und baue das Projekt neu.

### Aufgabe 4.11 Variablen in make

Im Moment sieht unser Makefile in etwa so aus:

```
all: main.exe

main.exe: main.o Building.o
   g++ -o main.exe main.o Building.o

%.o: %.cpp
   g++ -MMD -MP -c $< -o $@

-include Building.d main.d

clean:
   rm -rf main.o Building.o main.d Building.d main.exe

.PHONY: clean all
```

Dir ist sicherlich eine andere Form der Redundanz aufgefallen: Noch immer haben wir die Tatsache, dass es im Moment zwei Sourcedateien gibt, an unterschiedlichen Stellen im Makefile festgelegt. Wenn wir nun als nächstes die `Floor`-Klasse entwerfen, müssten wir diese hinzufügen

- als Abhängigkeit von `main.exe`,
- zum Linker-Aufruf in `main.exe`,
- in der `-include`-Direktive und

- im Target `clean`, und das gleich doppelt!

Das ist natürlich immer noch ziemlich fehleranfällig.
Wir würden also gerne nur an *einer* Stelle definieren, welche Sourcedateien Teil unseres Projektes sind.
Da die Sourcedateien nirgendwo im Makefile auftreten, fangen wir mit den Object-Dateien an. Lege am Anfang des Makefiles eine Variable mit dem Inhalt '`main.o Building.o`' und ersetze das Auftreten der beiden Object-Dateien mit dieser Variablen:

```
OBJECTS=main.o Building.o

all: main.exe

main.exe: $(OBJECTS)
   g++ -o main.exe $(OBJECTS)

# and so on...
```

Wiederhole die Prozedur für die Dependency-Dateien ('`DEPEND=main.d Building.d`') und das ausführbare Programm ('`BINARY=main.exe`'). Lasse dein Programm zwischendurch immer wieder vollständig neu bauen, um sicherzustellen, dass nichts kaputt geht.
Am Ende sollte dein Makefile in etwa so aussehen:

```
BINARY=main.exe
OBJECTS=main.o Building.o
DEPEND=main.d Building.d

all: $(BINARY)

$(BINARY): $(OBJECTS)
   g++ -o $(BINARY) $(OBJECTS)

%.o: %.cpp
   g++ -MMD -MP -c $< -o $@

-include $(DEPEND)

clean:
   rm -rf $(OBJECTS) $(DEPEND) $(BINARY)

.PHONY: clean all
```

Wie wir die verbliebene Redundanz auflösen, sehen wir in der nächsten Teilaufgabe.

### Aufgabe 4.12 Wildcard-Ausdrücke

Die beiden Variablen `OBJECTS` und `DEPEND` sind strukturell ähnlich – wieder etwas, das wir loswerden wollen. Außerdem wäre es doch viel schöner, an einer Stelle die Sourcedateien zu definieren, oder?
Lege dazu eine neue Variable *SOURCES* mit den beiden Sourcedateien an. Der folgende Snippet zeigt, wie man nun mittels Suffix-Ausdrücken die anderen beiden Variablen erzeugt:

```
BINARY   = main.exe
SOURCES  = main.cpp Building.cpp
OBJECTS  = $(patsubst %.cpp, %.o, $(SOURCES))
DEPEND   = $(patsubst %.cpp, %.d, $(SOURCES))
```

Es geht sogar noch allgemeiner: Du kannst per regulärem Ausdruck definieren, dass *alle* Sourcedateien im aktuellen Ordner verwendet werden sollen, indem du *SOURCES* wie folgt definierst:

```
SOURCES=$(wildcard ./*.cpp)
```

## Zusatzaufgaben

### Aufgabe 4.13 Zusammenfassung

Das Produkt unserer Bemühungen in dieser Aufgabe ist ein Makefile, das unabhängig davon ist, wie viele Sourcedateien du im aktuellen Verzeichnis hältst und wie sie genau heißen – wichtig ist nur die Endung *cpp*.
Hier nochmal das vollständige Makefile:

```
BINARY   = main.exe
SOURCES  = main.cpp Building.cpp
OBJECTS  = $(patsubst %.cpp, %.o, $(SOURCES))
DEPEND   = $(patsubst %.cpp, %.d, $(SOURCES))

all: $(BINARY)

$(BINARY): $(OBJECTS)
   g++ -o $(BINARY) $(OBJECTS)

%.o: %.cpp
   g++ -MMD -MP -c $< -o $@

-include $(DEPEND)

clean:
   rm -rf $(OBJECTS) $(DEPEND) $(BINARY)

.PHONY: clean
```

### Aufgabe 4.14 Nachwort

Dies hier sind nur äußerst wenige der Möglichkeiten, die `make` bietet[3].
In der Praxis existieren Build-Tools, die eine wesentlich besser zu verstehende Beschreibungssprache verwenden und daraus Makefiles generieren. Beispiele sind `cmake`[4] oder `qmake`[5] (Bestandteil von Qt).

[3] Für einen besseren Eindruck, sieh dir die Doku an: `https://www.gnu.org/software/make/manual/html_node/index.html`
[4] `http://www.cmake.org/cmake/help/cmake_tutorial.html`
[5] `http://qt-project.org/doc/qt-4.8/qmake-tutorial.html`