Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 1. Tag

Einführung

Für alle Übungen des C/C++ Praktikums wird Eclipse zusammen mit dem C/C++ Development Tooling (CDT) und dem *GNU project C and C++ compiler* (GCC) verwendet. Wir gehen davon aus, dass der generelle Umgang mit Eclipse bereits aus der Java-Programmierung bekannt ist.
Die Materialien zur Vorlesung und Übung sind in zwei Git-Repositories zu finden. Diese beinhalten zum einen die Folien und die Programmbeispiele aus der Vorlesung und zum anderen diese Übungsblätter, Vorlagen für die letzten beiden Tage des Praktikums und alle Lösungen zu den Übungsaufgaben.

- **Vorlesung:** `https://github.com/Echtzeitsysteme/tud-cpp-lecture`
- **Übungen/Git- und Eclipseeinführung:** `https://github.com/Echtzeitsysteme/tud-cpp-exercises`

Weiterhin findest du im Übungsrepository eine kleine Einführung in Git in Eclipse. Es ist ausdrücklich empfohlen sich sowohl die Unterlagen mit Git herunterzuladen, als auch Git lokal für die eigenen Übungsaufgaben zu verwenden. Ein Git-Plugin ist in neueren Eclipseversionen standardmäßig verfügbar, das zum Beispiel über **Window → Open Perspective... → Git** geöffnet verwendet werden kann.

Hinweise

- Bei Fragen und Problemen aktiv um Hilfe bitten!
- Alle Lösungen enthalten ein Makefile und können entweder über die Kommandozeile mit Hilfe von `make` kompiliert werden, oder in Eclipse, indem du sie als Makefileprojekt importierst. Achte darauf, beim ersten Ausführen innerhalb von Eclipse `gdb/mi` (ein Interface für den Debugger `gdb`) auszuwählen.
- Folgende Eclipse Shortcuts könnten sich dir im Verlauf des Praktikums als nützlich erweisen:

Ctrl+Space	Autocomplete	Anzeige von Vervollständigungshinweisen (z.B. nach `std::` oder `main`)
Alt+Shift+R	Rename	Umbennenen von Variablen, Funktionen, Klassen, ...
Ctrl+N	New	Anlegen neuer Ressourcen (Dateien, Projekte, ...)
Ctrl+Tab	Header↔Source	Wechsel zwischen der Header- und der Implementierungsdatei
Ctrl+Click/F3	Go to	Navigiert zu der Definition eines Elements (Funktion, Klasse, Variable, ...)
Ctrl+B	Build	Startet den Buildprozess (Aufruf von Compiler und Linker)

Aufgabe 0 Hello World

Lege ein neues C++ Projekt an, indem du **File → New → C++ Project** im Eclipse Menü wählst und als Projekttyp **Empty Project** auswählst. Füge eine neue Sourcedatei zum Projekt hinzu, indem du mit der rechten Maustaste auf das Projekt klickst und **New → Source File** auswählst. Gib als Dateinamen `main.cpp` ein und bestätige mit **Finish**. Füge folgendes Programm ein, kompiliere es mit Hilfe des Build-Symbols („Hammer“) und führe es aus.

```
#include <iostream>
int main() {
   std::cout << "Hello World" << std::endl;  // prints "Hello World"
}
```

Jedes vollständige C++ Programm muss **genau eine** Funktion mit Namen `main` und Rückgabetyp `int` außerhalb von Klassen im globalen Namensraum besitzen. Andernfalls wird der Linker mit der Fehlermeldung *undefined reference to*

'main' abbrechen. Der Rückgabetyp wird verwendet um dem Aufrufer (Betriebssystem, Shell, ...) den Erfolg oder Misserfolg der Ausführung zu signalisieren. Typischerweise wird im Erfolgsfall 0 zurückgegeben.
Die erste Zeile des obigen Programms bindet den Header der `iostream` Bibliothek ein, welche unter anderem Klassen und Funktionen zur Ein- und Ausgabe mit Hilfe von << (*insertion operator*) und >> (*extraction operator*) anbietet. Diese Bibliothek ist Teil der C++-Standardbibliothek, welche eine Sammlung an generischen Containern, Algorithmen und vielen häufig genutzten Funktionen ist. Um auf die Elemente dieser Bibliothek zuzugreifen, muss man ihren `namespace` (in diesem Fall `std`) voranstellen, gefolgt von zwei Doppelpunkten und dem gewünschten Element (in diesem Fall `cout` und `endl`). Um Überschneidungen mit eigenen Definitionen zu vermeiden, ist es üblich Bibliotheken in einem `namespace` zu kapseln, welcher analog zu `package` in Java funktioniert, jedoch nicht an Ordnerstrukturen gebunden ist.
In der dritten Zeile wird der String `"Hello World"` in `std::cout` eingefügt, gefolgt von `std::endl`, das einen Zeilenumbruch erzeugt und die Ausgabepuffer leert. Für weitere Informationen zur Kommandozeilenausgabe siehe `http://www.cplusplus.com/doc/tutorial/basic_io/` und `http://www.cplusplus.com/reference/iomanip/`.

Hinweise

- Einzeilige Kommentare können durch `//`, mehrzeilige durch `/* ... */` eingeschlossen werden.
- Anders als in Java können Funktionen auch außerhalb von Klassen definiert und verwendet werden.
- Die `return` Anweisung darf in der `main` Funktion weggelassen werden.

Häufige (Compiler-)Fehlermeldungen

Im Folgenden sind einige Fehlermeldungen des `gcc` zusammen mit möglichen Lösungsstrategien aufgelistet. Die generelle Faustregel lautet: **Compilerfehler sollten immer von oben nach unten abgearbeitet werden, so wie sie in der Konsole erscheinen.** Der Grund hierfür ist, dass es durch einen Fehler zu weiteren Folgefehlern kommen kann.

`Launching failed: Binary not found`

> Dieser Fehler wird von Eclipse geworfen, wenn es nach dem Kompilieren das lauffähige Programm nicht findet. Hier sollte man sicherstellen, dass der `Elf Parser` (oder `PE Windows Parser` unter Windows) aktiv ist: `Window -> Preferences -> C/C++Build / Settings -> Tab: Binary Parsers -> Elf Parser`. Alternativ kann man in der `Run Configuration` die auszuführende Datei manuell eintragen.

`error: expected ';' before ...`

> Dies bedeutet, dass in der Zeile davor ein **;** vergessen wurde. Allgemein beziehen sich Fehlermeldungen **expected ... before ...** häufig auf die Zeile **vor** dem markierten Statement. Beachte, dass *die Zeile davor* auch die letzte Zeile einer eingebundenen Header-Datei sein kann. Beispiel:
>
> ```
> #include "main.h"
> int main() {
>    ...
> }
> ```
>
> Falls im Header `main.h` in der letzten Zeile ein Semikolon fehlt, wird der Compiler die Fehlermeldung trotzdem auf die Zeile `int main() {` beziehen!!

`error: invalid conversion from <A> to <B>.`

> Dies bedeutet, dass der Compiler an der entsprechenden Stelle einen Ausdruck vom Typ *B* erwartet, im Code jedoch ein Ausdruck vom Typ *A* angegeben wurde. Insbesondere bei verschachtelten Typen sowie (später vorgestellten) Zeigern und Templates kann die Fehlermeldung sehr lang werden. In so einem Fall lohnt es sich, den Ausdruck in mehrere Teilausdrücke aufzubrechen und die Teilergebnisse durch temporäre Variablen weiterzureichen.

`undefined reference to ...`

> Dies bedeutet, dass das Programm zwar korrekt kompiliert wurde, der Linker aber die Definition des entsprechenden Bezeichners nicht finden kann. Das kann passieren, wenn man dem Compiler durch einen Prototypen mitteilt, dass eine bestimmte Funktion existiert (**deklariert**), diese aber nirgendwo tatsächlich **definiert**. Überprüfe in diesem Fall, ob der Bezeichner tatsächlich definiert wurde und ob die Signatur der Definition mit dem Prototypen übereinstimmt.

Übung zum C/C++-Praktikum - Tag 1

Aufgabe 1 C++ Grundlagen, Funktionen und Strukturierung

Für diese Aufgabe kannst du entweder das vorherige Programm weiter entwickeln oder genauso wie vorher ein neues Projekt anlegen.

Exkurs Primitive Datentypen

Die primitiven Datentypen in C++ sind ähnlich denen in Java. Allerdings sind alle Ganzzahl-Typen in C++ sowohl mit als auch ohne Vorzeichen verfügbar. Standardmäßig sind Zahlen vorzeichenbehaftet. Mittels `unsigned` kann man vorzeichenlose Variablen deklarieren. Durch das freie Vorzeichenbit kann ein größerer positiver Wertebereich dargestellt werden.

```
int i;                  // signed int, -2147483648 to +2147483647 on a 32-bit machine
unsigned int ui;        // unsigned int, 0 to 4294967295 on a 32-bit machine
// unsigned double d;   // not possible
```

Eine andere Besonderheit von C++ ist, dass Ganzzahlwerte implizit in Boolesche Werte (Typ: `bool`) umgewandelt werden. Alles ungleich 0 wird als `true` gewertet, 0 als `false`. Somit können Ganzzahlen direkt in Bedingungen ausgewertet werden.

Aufgabe 1.1 Sterne

Schreibe eine Funktion `printStars(int n)`, die `n`-mal ein * auf der Konsole ausgibt und mit einem Zeilenumbruch abschließt. Ein Aufruf von `printStars(5)` sollte folgende Ausgabe generieren:

```
*****
```

Platziere die Funktion **vor** der `main`, da sie sonst von dort aus nicht aufgerufen werden kann. Benutze die erstellte Funktion `printStars(int n)`, um eine weitere Funktion zu schreiben, die eine Figur wie unten dargestellt ausgibt. Verwende hierzu Schleifen.

```
*****
****
***
**
*
**
***
****
*****
```

Hinweise

- Was die Benennung von Funktionen, Variablen und Klassen angeht, bist du frei. Für Klassen ist CamelCase wie in Java üblich. Bei Funktionen und Variablen wird zumeist entweder auch Camel Case oder Kleinschreibung mit Unterstrichen verwendet.

Aufgabe 1.2 Auslagern der Datei

Erstelle eine neue Header-Datei `functions.hpp` und eine neue Sourcedatei `functions.cpp`. Klicke hierzu mit der rechten Maustaste auf das Projekt und wähle **New → Source File**, bzw. wähle **New → Header File** und bestätige beide Dialoge mit **Finish**. Beachte hierbei, dass Eclipse automatisch Include-Guards in der Headerdatei erzeugt, die das mehrfache Einbinden desselben Headers verhindern:

```
#ifndef FUNCTIONS_HPP_
#define FUNCTIONS_HPP_
// your header ...
#endif /* FUNCTIONS_HPP_ */
```

Binde danach `functions.hpp` in beide Sourcedateien ein indem du

```
#include "functions.hpp"
```

verwendest. Verschiebe deine beiden Funktionen nach `functions.cpp`.
Schreibe nun in `functions.hpp` **Funktionsprototypen** für die beiden Funktionen aus der vorherigen Aufgabe. Funktionsprototypen dienen dazu, dem Compiler mitzuteilen, dass eine Funktion mit bestimmtem Namen, Parametern und Rückgabewert existiert. Ein Prototyp ist im wesentlichen eine mit `;` abgeschlossene Signatur der Funktion ohne Funktionsrumpf. Der Prototyp von `printStars(int n)` lautet `void printStars(int n);`
Fertig – die Ausgabe des Programms sollte sich nicht verändert haben.

Hinweise

- Sourcedateien tragen in der Regel die Endung *.cpp*, Headerdateien *.h* oder *.hpp*.
- Denke daran, auch in `functions.cpp` `iostream` einzubinden, falls du dort Ein- und Ausgaben verwenden willst.
- Beachte, dass es zwei verschiedene Möglichkeiten gibt, eine Header-Datei einzubinden - per `#include <Bibliotheksname>` sowie per `#include "Dateiname"`. Bei der ersten Variante sucht der Compiler nur in den Include-Verzeichnissen der Compiler-Toolchain, während bei der zweiten Variante auch die Projektordner durchsucht werden. Somit eignet sich die erste Schreibweise für System-Header und die zweite für eigene, projektspezifische Header.
- Anstelle der Include-Guards kannst du auch die Präprozessordirektive `#pragma once` verwenden. Dies wird von den meisten Compilern unterstützt.

Aufgabe 1.3 Eingabe

Erweitere das Programm um eine Eingabeaufforderung zur Bestimmung der Breite der auszugebenden Figur. Die Breite soll dabei eine im Programmcode vorgegebene Grenze nicht überschreiten dürfen. Gib gegebenenfalls eine Fehlermeldung aus. Verwende zum Einlesen `std::cin` und den `operator»`.

```
int zahl;
std::cin >> zahl;
```

Erstelle auch für diesen Aufgabenteil eine eigene Funktion und lagere diese nach `functions.cpp` aus.

Aufgabe 1.4 Fortlaufendes Alphabet

Statt eines einzelnen Zeichens soll nun das fortlaufende Alphabet ausgegeben werden. Sobald das Ende des Alphabets erreicht wurde, beginnt die Ausgabe erneut bei *a*. Beispiel:

```
abc
de
f
gh
ijk
```

Implementiere dazu eine Funktion `char nextChar()`. Diese soll bei jedem Aufruf das nächste auszugebende Zeichen von Typ `char` zurückgeben, beginnend bei *a*. Dazu muss sich `nextChar()` intern das aktuelle Zeichen merken. Dies kann durch die Verwendung von statischen Variablen erreicht werden. Diese behalten ihren alten Wert beim Wiedereintritt in die Funktion. Ein statische Variable `c` wird mittels

```
static char c = 'a';
```

deklariert. In diesem Fall wird `c` **einmalig zu Beginn des Programms** mit *a* initialisiert und kann später beliebig verändert werden.

Hinweise

- Der Datentyp `char` kann wie eine Zahl verwendet werden, d.h. man kann z.B. die Modulooperation % verwenden.

Übung zum C/C++-Praktikum - Tag 1

Aufgabe 1.5 Namensräume

Sinnvollerweise werden Bibliotheken in einen eigenen Namensraum gekapselt, damit ihre Funktionen nicht zufällig mit selbstgeschriebenen Funktionen kollidieren. Erweitere dazu das Programm, indem du im Header die Funktionsprototypen wie folgt in einen `namespace` setzt.

```
namespace fun {
   // function prototypes ...
}; // semicolon!
```

Denke daran, dass du die Namen der Funktionen in der Sourcedatei noch anpassen musst, indem du vor jede Funktion den gewählten `namespace`-Namen gefolgt von zwei Doppelpunkten setzt. Genauso muss der Namensraum vor jedem Aufruf der Funktion gesetzt werden.

```
void fun::print_star(int n) {
   // ...
}
```

Vergisst man den Namensraum in der Sourcedatei anzugeben, findet der Linker keine Implementation zu der im Header definierten Funktion. Weiterhin stünde diese Funktion nicht mehr im Bezug zum Header und könnte nur noch lokal verwendet werden (`print_star(int n)` und `fun::print_star(int n)` sind unterschiedliche Funktionen!).

Hinweise

- Du kannst `using namespace fun;` verwenden, um diesen Namensraum zu importieren (vergleichbar mit `static import` in Java). Genauso wie in Java kann es hierdurch leichter zu Namenskollisionen kommen und sollte daher eher nicht verwendet werden.

Übung zum C/C++-Praktikum - Tag 1

Aufgabe 2 Klassen

Ziel dieser Aufgabe ist es die vorherige Aufgabe objektorientiert zu lösen. Schreibe hierfür manuell eine Klasse, die das aktuelle Zeichen als Attribut enthält und durch Methoden ausgelesen und inkrementiert werden kann.

Hinweise

- Verwende in dieser Aufgabe noch **nicht** den Klassengenerator (**New** → **Class** im Kontext-Menü des Projekts) von Eclipse!

Aufgabe 2.1 Definition

Eine Klasse wird üblicherweise analog zu der vorherigen Aufgabe in Deklaration (Headerdatei) und Implementation (Sourcedatei) aufgeteilt. Die Struktur der Klasse mit allen Attributen und Funktionsprototypen wird im Header beschrieben, während die Sourcedatei nur die Implementation der Funktionen und Initialisierungen statischer Variablen enthält. Standardmäßig sind alle Elemente einer Klasse privat. Im Gegensatz zu Java werden in C++ die Access-Modifier `public/private/protected` nicht bei jedem Element einzeln sondern blockweise angegeben.

```
class ClassName {
public:
   // public members ...
private:
   // private members ...
}; // semicolon!
```

Erzeuge einen Header `CharGenerator.hpp` und erstelle den Klassenrumpf der `CharGenerator` Klasse. Füge der Klasse das `private` Attribut `char nextChar` hinzu, in dem das als nächstes auszugebende Zeichen gespeichert wird und einen `public` Konstruktorprototypen `CharGenerator()`, der `nextChar` auf a initialiseren soll. Füge noch eine `public` Funktionsprototypen `char generateNextChar()` hinzu, welche das nächste auszugebende Zeichen zurückgeben soll.

Hinweise

- Ein Konstruktor wird als eine Funktion ohne Rückgabetyp deklariert, die den gleichen Namen wie die Klasse hat, und beliebige Parameter beinhalten kann.

Aufgabe 2.2 Implementation

Wie bei der Verwendung von `namespace` muss der Scope der Klasse (der Klassenname) in der Sourcedatei vor jeder Elementbezeichnung (Konstruktor, Funktion, ...) durch zwei Doppelpunkte getrennt angegeben werden.

```
void ClassName::functionName() {
   // function implementation ...
}
```

Um Attribute zu initialisieren, wird üblicherweise eine sogenannte Initialisierungsliste im Konstruktor verwendet, da diese vor dem Eintritt in den Konstruktorrumpf aufgerufen wird. Die Initialisierungsliste wird durch einen Doppelpunkt zwischen der schließenden Klammer der Parameterliste und der geschweiften Klammer des Rumpfes eingeleitet, und bildet eine mit Komma separierte Liste von Attributnamen und ihren Initialisierungsargumenten in Klammern.

```
ClassName::ClassName():
   // initializer list:
   attributeOne(initialValueOne),
   attributeTwo(initialValueTwo)
{
   // constructor body
}
```

Erzeuge eine Sourcedatei `CharGenerator.cpp` für die Implementation der Klasse und binde die `CharGenerator.hpp` ein. Implementiere den Konstruktor, indem du `nextChar` mit a in der Initialisierungsliste initialisierst. Implementiere zudem `generateNextChar()`, indem du `nextChar` zurückgibst.

Hinweise

- Die Reihenfolge der Initialisierungsliste sollte der Deklarationsreihenfolge entsprechen.
- Konstanten **müssen** in der Initialisierungsliste zugewiesen werden, damit diese zur Laufzeit bekannt sind.

Aufgabe 2.3 Instanziierung

Erzeuge wie aus den vorherigen Aufgaben bekannt eine `main.cpp` mit einer `main()` Funktion in der du ein `CharGenerator`-Objekt erzeugst und `generateNextChar()` mehrfach aufrufst und ausgibst.

```
CharGenerator charGen;
char next = charGen.generateNextChar();
std::cout << next << std::endl;
```

Überprüfe das Ergebnis über die Konsole oder den Debugger.

Hinweise

- Um ein Objekt zu erzeugen, muss in C++ kein `new` verwendet werden (siehe dazu nächste Vorlesung).

Aufgabe 2.4 Default-Parameter

Damit man nicht immer das Startzeichen angeben muss, kann man sogenannte Default-Parameter angeben, der beim Aufruf weggelassen werden kann. Hierzu wird dem Parameter im Prototypen (im Header) ein Wert zugewiesen ohne die Implementation zu ändern.

```
class CharGenerator {
public:
   CharGenerator(char initialChar = 'a');
   //...
};
```

Erweitere den Konstruktor um einen Parameter `char initialChar`, welcher defaultmäßig *a* ist und ändere die Initialisierung von `nextChar`, damit dieser mit dem übergebenen Parameter gestartet wird.
Teste deine Implementation sowohl mit als auch ohne Angabe des Startzeichens. Um ein Startzeichen anzugeben, lege das Objekt wie folgt an:

```
CharGenerator charGen('x');
```

Hinweise

- Bei der Definition eines Default-Parameters müssen für alle nachfolgenden Parameter ebenfalls Default-Werte angegeben werden, um Mehrdeutigkeiten beim Aufruf zu vermeiden.

Aufgabe 2.5 PatternPrinter

Implementiere folgende Klasse.

```
class PatternPrinter {
public:
   PatternPrinter();
   void printPattern();       // read width and print chars in a pattern
private:
   CharGenerator charGen;
   void printNChars(int n);   // print n characters to the console
   int readWidth();           // read width (user input)
};
```

Teste deine Implementation, indem du ein `PatternPrinter`-Objekt anlegst und `printPattern()` darauf aufrufst.

Hinweise

- Ohne eine Initialisierungsliste wird `charGenerator` mit dem Default-Parameter initialisiert. Um ein eigenes Startzeichen anzugeben, muss eine Initialisierungsliste erstellt und `charGenerator` mit dem entsprechenden Argument initialisiert werden.

Übung zum C/C++-Praktikum - Tag 1

Aufgabe 3 Operatorüberladung

In C++ besteht die Möglichkeit, Operatoren wie `+` (`operator+`), `*` (`operator*`),... zu überladen. Man kann selber spezifizieren, was beim Verknüpfen von Objekten mit einem Operator geschehen soll, um zum Beispiel den Quellcode übersichtlicher zu gestalten. Du hast bereits das Objekt `std::cout` der Klasse `ostream` kennengelernt, welche den `<<`-Operator überlädt, um Ausgaben von `std::string`, `int`,... komfortabel zu tätigen. In dieser Aufgabe sollst dueine eigene Vektor-Klasse schreiben und einige Operatoren überladen.

Hinweise

- Am Tag 4 soll dieser Vektor um weitere Funktionen erweitern werden. Falls du mit dieser Aufgabe bis dahin nicht fertig sein solltest, kannst du natürlich auf die Musterlösung zurückgreifen.

Aufgabe 3.1 Konstruktor und Destruktor

Implementiere folgende Klasse. Füge jedem Konstruktor und Destruktor eine Ausgabe auf der Konsole hinzu, um beim Programmlauf den Lebenszyklus der Objekte nachvollziehen zu können.

```
class Vector3 {
public:
   Vector3();                                // initialize vector with zero
   Vector3(double a, double b, double c);    // initialize vector with a, b, c
   Vector3(const Vector3& other);            // copy constructor: copy a vector
   ~Vector3();                               // destructor: destroy the vector
private:
   double a, b, c;                           // vector components
};
```

Der Copy-Konstruktor wird aufgerufen, wenn das Objekt kopiert werden soll, z.B. für eine Call-by-Value Parameterübergabe. Jeder Copy-Konstruktor benötigt eine Referenz auf ein Objekt vom gleichen Typ wie die Klasse selbst als Parameter. Sinnvollerweise wird noch `const` vor oder nach der Typbezeichnung eingefügt (aber vor `&`), da typischerweise das Ursprungsobjekt nicht verändert wird.
Der Destruktor wird aufgerufen, sobald die Lebenszeit eines Objekts endet. Er wird verwendet, um Ressourcen die das Objekt besitzt freizugeben. Die Syntax des Prototypen lautet

```
~ClassName();
```

und die Implementation entsprechend

```
ClassName::~ClassName() {/* destructor implementation ... */}
```

Hinweise

- Es dürfen eine beliebige Anzahl an Konstruktoren mit verschiedenen Parametersätzen existieren.
- Der Compiler wird automatisch einen `public` Destruktor und `public` Copy-Konstruktor erzeugen, falls sie nicht **deklariert** wurden. Ebenso wird ein `public` Defaultkonstruktor (keine Argumente) automatisch vom Compiler generiert, falls überhaupt keine Konstruktoren deklariert wurden.
- Würden beim Copy-Konstruktor `other` by-Value übergeben werden, müsste eine Kopie von `other` angelegt werden. Dazu würde der Copy-Konstruktor aufgerufen, was zu einer unendlichen Rekursion führt, bis der Stack seine maximale Größe überschreitet und das Programm abstürzt.

Aufgabe 3.2 Vektoraddition, -subtraktion und Skalarprodukt

Erweitere die Klasse um folgende `public` Funktionen, um Vektoren durch `v1 + v2`, `v1 - v2` und `v1 * v2` addieren/subtrahieren und das Skalarprodukt bilden zu können, indem die Operatoren +, - und * überladen werden.

```
Vector3 operator+(Vector3 rhs);     // add two vectors component-by-component
Vector3 operator-(Vector3 rhs);     // subtract two vectors component-by-component
double operator*(Vector3 rhs);      // determine the dot product of two vectors
```

Innerhalb der Methode kannst du durch `a`, `b` und `c` auf eigene Attribute und über `rhs.a`, `rhs.b` und `rhs.c` auf Attribute der rechten Seite zugreifen. Denke daran, bei der Implementation den Klassen den Scope der Klasse in der Sourcedatei vor jeder Elementbezeichnung durch zwei Doppelpunkte getrennt anzugeben.

```
Vector3 Vector3::operator+(Vector3 rhs) {/* function implementation ...*/}
Vector3 Vector3::operator-(Vector3 rhs) {/* function implementation ...*/}
double Vector3::operator*(Vector3 rhs)  {/* function implementation ...*/}
```

Hinweise

- Der Parameter `rhs` steht für die rechte Seite („right-hand-side“) des jeweiligen Operators. Dadurch, dass der Operator als Member der Klasse deklariert wurde, nimmt die aktuelle Instanz hierbei automatisch die linke Seite der Operation an.
- Der Rückgabetyp eines Skalarprodukts (dot product) ist kein `Vector3` sondern ein Skalar (`double`)!

Aufgabe 3.3 Ausgabe

Überlade den << Operator zur Ausgabe eines Vektors mit der gewohnten `std::cout << ...` Syntax, indem du den folgenden Funktionsprototypen **außerhalb** der Klassendefinition setzt

```
std::ostream& operator<<(std::ostream &out, Vector3 rhs);
```

und innerhalb der Sourcedatei wie folgt implementierst.

```
std::ostream& operator<<(std::ostream &out, Vector3 rhs)  {
   out << ... ;
   return out;
}
```

Definiere zudem Funktionen die die Werte für die `private` Attribute `a`, `b` und `c` zurückgeben (Getterfunktionen), anderenfalls wird der Compiler beim Zugriff auf die Attribute folgende Fehlermeldung werfen.

```
error: 'double Vector3::a' is private within this context
```

Hinweise

- Denke daran, den `iostream` Header einzubinden.
- Diesmal musste die Überladung **außerhalb** der `Vektor3`-Klasse definiert werden, weil das `Vektor3`-Objekt auf der rechten Seite der Operation steht. Als linke Seite wird hierbei ein `ostream`-Objekt (wie z.B. `std::cout`) erwartet, um Ausgabeketten `std::cout << ... << ...` zu ermöglichen. Hierzu muss das Ausgabeobjekt auch zurückgegeben werden, damit das `ostream`-Objekt aber nicht jedes Mal kopiert wird, wird es als Referenz **&** durchgereicht.
- Anstatt Getter und Setter für `private` Attribute zu schreiben, kann man auch einer Klasse oder Funktion vollen Zugriff mit Hilfe des Schlüsselworts `friend` erlauben. In der nächsten Übung wird hierauf noch einmal eingegangen.

Aufgabe 3.4 Testen

Teste deine bisher definierten Methoden und Funktionen. Probiere auch Kombinationen von verschiedenen Operatoren aus und beobachte das Ergebnis. Schreibe auch eine einfache Funktion, die Vektoren als Parameter nimmt. Wie du siehst, werden sehr viele `Vector3`-Objekte erstellt, kopiert und gelöscht. Dies liegt daran, dass die Objekte immer per Call-By-Value übergeben und dabei kopiert werden. Wie dies vermieden werden kann, siehst du im nächsten Teil des Praktikums.

Übung zum C/C++-Praktikum - Tag 1

Aufgabe 4 Aufzugsimulator

In dieser Aufgabe soll ein Grundgerüst für den in der Vorlesung vorgestellten Aufzugsimulator geschaffen werden. Diese Aufgabe wird in den folgenden Übungen erweitert. Falls du feststellst, dass dir im Moment die Zeit dafür fehlt, kannst du diese Aufgaben auch nach hinten verschieben und als ausführliche Klausurvorbereitung nutzen.

Aufgabe 4.1 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 4.2 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. 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 4.3 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 4.4 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 4.5 Komfortfunktionen

Implementiere nachfolgende `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*). 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 4.6 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.