Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 1. Tag

Generelle Hinweise

Die Instruktionen auf den Übungsblättern sind in der Du-Form gehalten; wir hoffen, dass ist kein Problem. :-)

C++ ist nicht einfach zu erlernen; insbesondere die Interpretation der Compilerfehler ist eine Kunst für sich. Bitte aktiv um Hilfe, wenn du nicht weiterkommst.

Shortcuts

Das Arbeiten mit Eclipse macht erst so richtig Spaß, wenn man einige Shortcuts beherrscht. Dazu zählen:

Autocomplete (*Ctrl+Space*) Hier erhältst du Vervollständigungsvorschläge und andere Hinweise. Versuche es mal mit *std::*, *using namespace*. Wenn du *main* tippst und *Ctrl+Space* drückst, generiert Eclipse für dich die *main*-Funktion.

Umbenennen (*Alt+Shift+R*) Mit dieser Funktion kannst du Variablen, Funktionen, Klassen,... umbenennen.

Neu (*Ctrl+N*) Hier kannst du schnell neue Resourcen (Dateien, Projekte,...) anlegen.

Header↔Source (*Ctrl+Tab*) Auf diese Weise kannst du schnell zwischen dem Header und der Implementierungdatei hin- und herspringen.

Go to (*Ctrl+Click*/*F3*) Wenn du *Ctrl* gedrückt hältst und auf ein Element klickst (Funktion, Klasse,...), navigiert Eclipse dorthin.

Aufgabe 1 Eclipse CDT

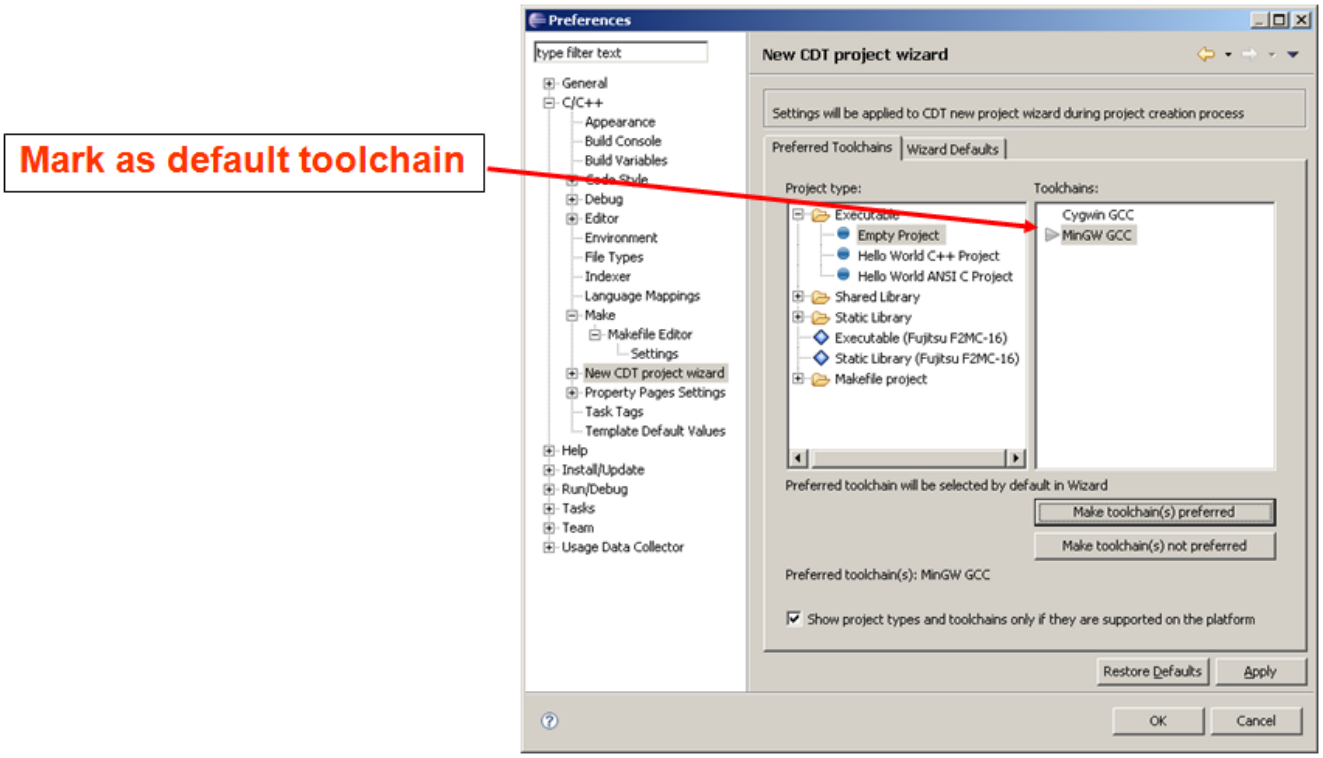
Für alle Übungen des C/C++ Praktikums werden wir Eclipse zusammen mit dem C/C++ Development Tooling (CDT) und dem MinGW gcc Compiler verwenden. Wir gehen davon aus, dass du den generellen Umgang Eclipse bereits aus der Java-Programmierung kennst und nur die Unterschiede zur C++-Programmierung erläutert werden müssen.

Alle Tools sind bereits auf den Poolrechnern vorinstalliert. Falls du mit deinem Nodebook arbeiten möchtest, findest du den Installer (Windows) unter: `https://www.dropbox.com/sh/rn714n1ugt6t3ke/AAB2rCt7FCHstSCp_NbnLekya?dl=0`.

Verwende zum Starten von Eclipse den ***Eclipse für CPPP → Eclipse*** Eintrag im Startmenü bzw. das ***eclipse.cmd*** Script im Installationsordner. Starte bitte nicht *eclipse.exe* direkt, weil dabei nicht alle benötigten Pfade gesetzt werden.

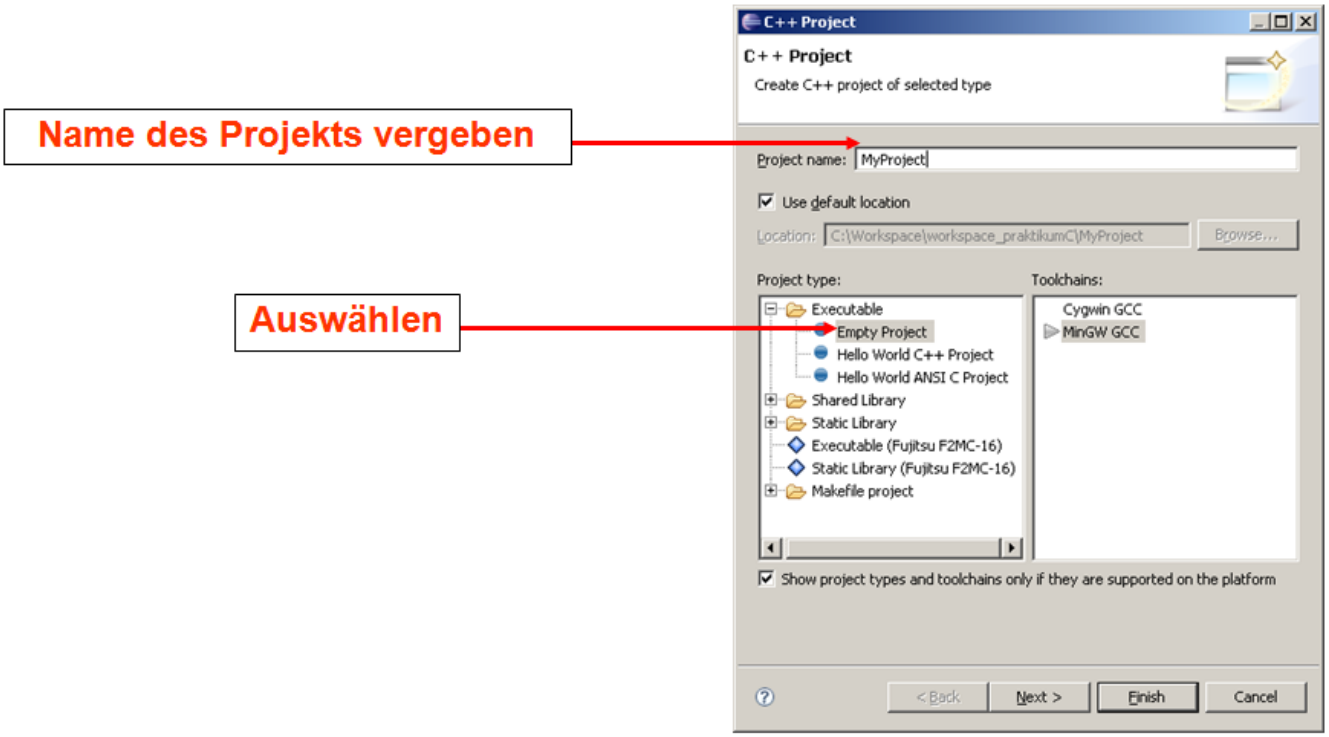
Aufgabe 1.1 Toolchain einstellen

Als erstes muss MinGW als die bevorzugte C/C++ Toolchain eingestellt werden. Öffne dazu den Menüpunkt **Window → Preferences** und wähle **C/C++ → New CDT project wizard**. Markiere nun **MinGW GCC** rechts in der Liste und klicke auf **Make toolchain(s) preferred**. Schließe den Dialog anschließend mit **OK**.



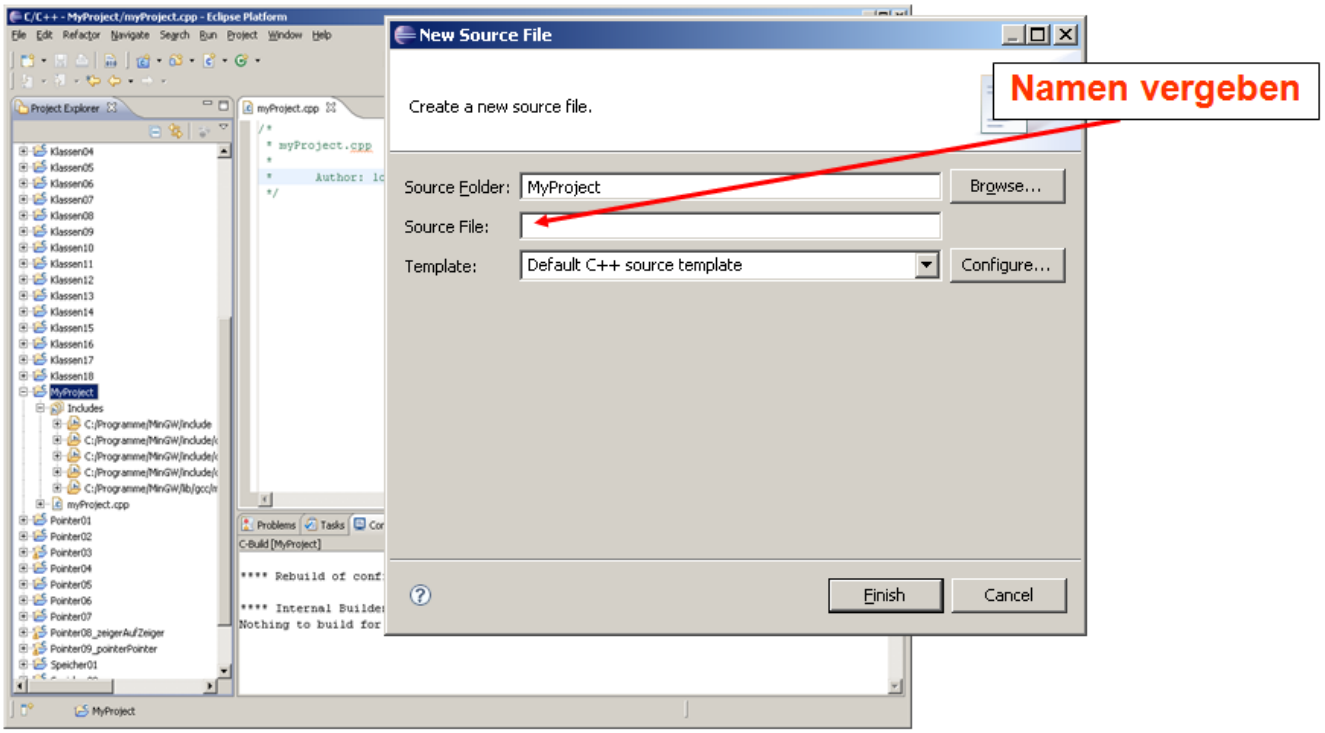
Aufgabe 1.2 Neues Projekt anlegen

Um ein neues Projekt anzulegen, wähle **File** → **New** → **C++ Project** im Eclipse Menü. Gib den gewünschten Projektnamen ein und wähle **Empty Project** bzw. **Hello World C++ Project** als Projekttyp aus. Die Toolchain sollte bereits automatisch auf MinGW voreingestellt sein.



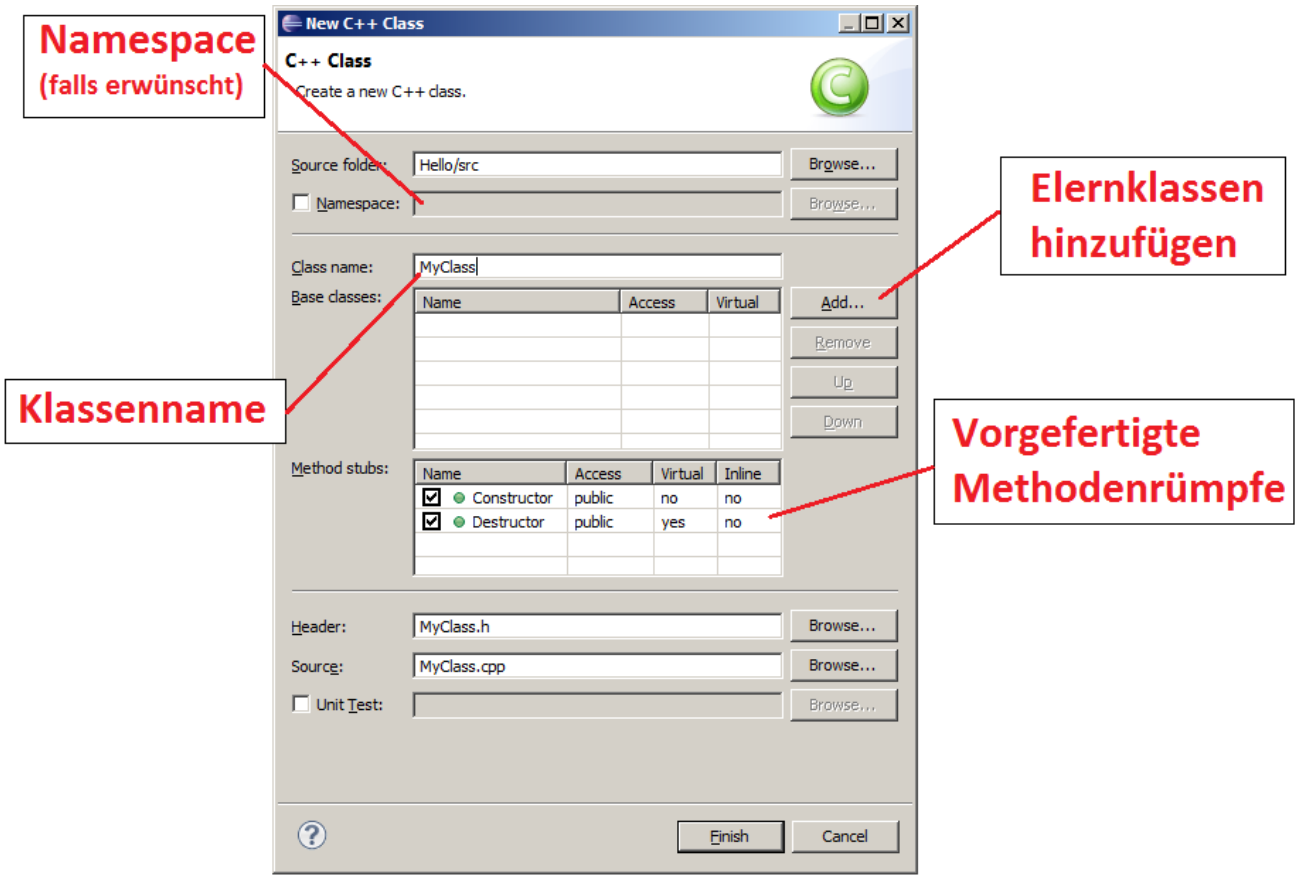
Aufgabe 1.3 Neue Dateien hinzufügen

Um eine neue Sourcecode-Datei zum Projekt hinzuzufügen, klicke mit der rechten Maustaste auf das Projekt und wähle **New → Source File**. Gib einen Dateinahmen (z.B. *main.cpp*) ein und bestätige mit **Finish**. Verfahrn analog, um Header-Dateien zu erstellen, wähle jedoch **New → Header File** im Kontextmenü. Sourcecode-Dateien tragen in der Regel die Endung *.cpp*, Header-Dateien *.h* oder *.hpp* (Wir empfehlen *.h* zu benutzen).



Aufgabe 1.4 Neue Klassen hinzufügen

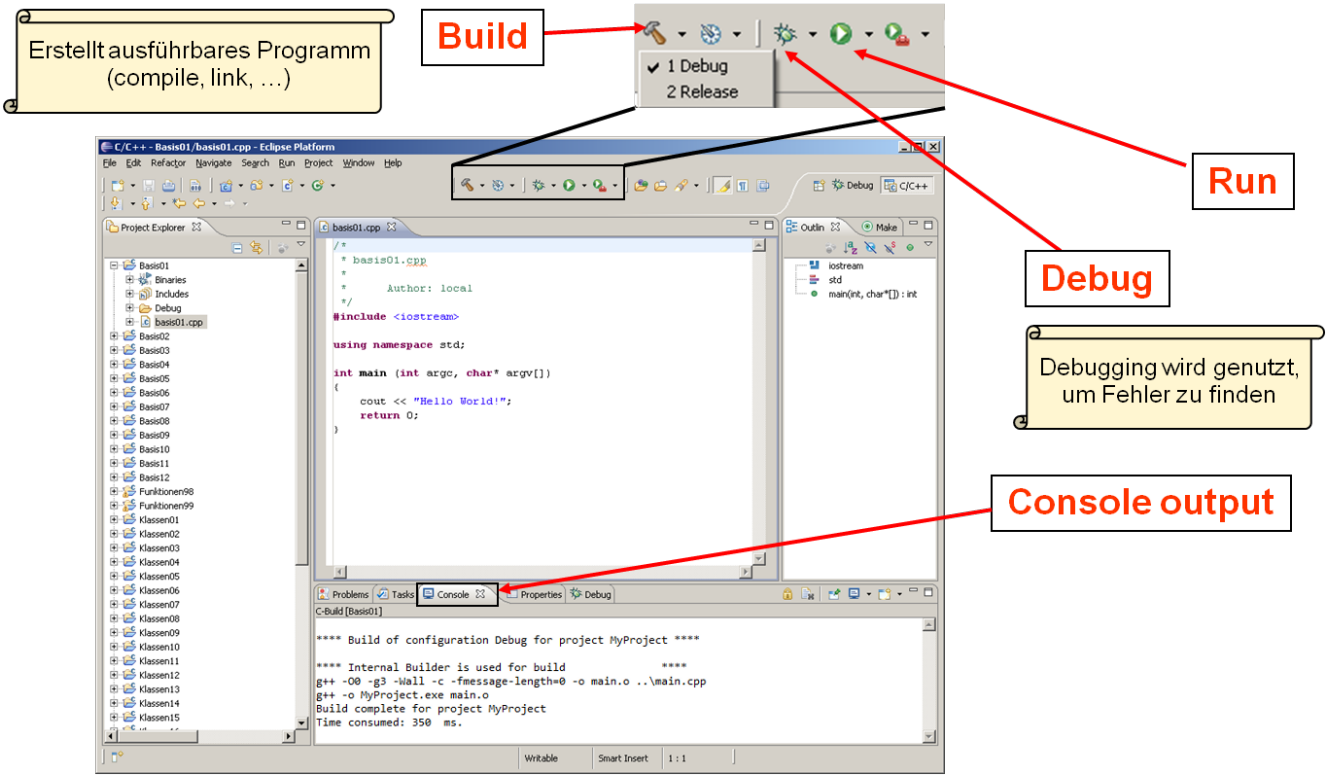
Für eine Klasse kann man Header- und Sourcecode-Datei in einem Schritt erzeugen. Wähle dazu **New → Class** im Kontext-Menü des Projekts, um den entsprechenden Wizard zu starten. Gib den Namen und bei Bedarf den Namespace sowie weitere Informationen wie z.B. die Elternklassen ein (dazu später mehr). Setze für die ersten Aufgaben den **virtual** Modifier des Destruktors auf **No**.



Aufgabe 1.5 Projekt kompilieren und starten

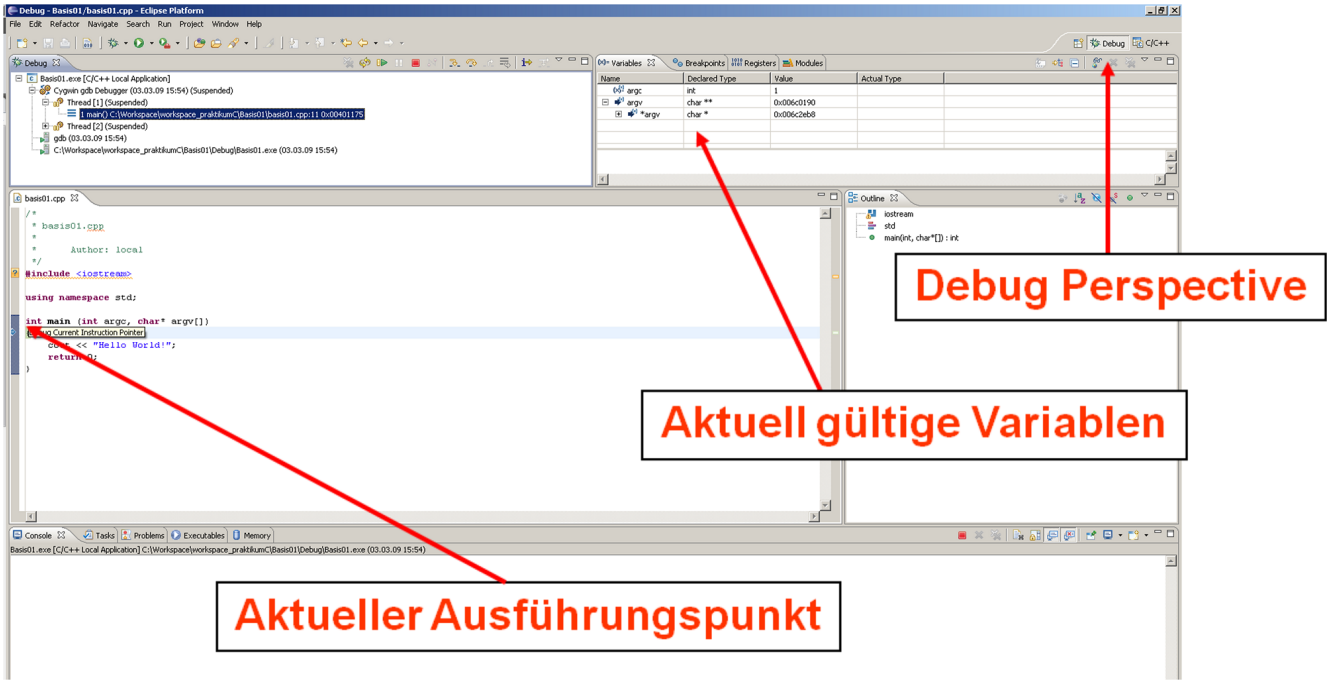
Bevor ein C++-Programm gestartet werden kann, muss es kompiliert werden. Im Gegensatz zu Java wird der Compiler nicht automatisch von Eclipse im Hintergrund aufgerufen, sondern muss manuell gestartet werden. Um das aktuell offene Projekt zu kompilieren, klicke auf das **Build**-Symbol („Hammer“) in der Eclipse C++ Toolbar. Im **Console**-Fenster unten werden Compiler-Meldungen und eventuelle Fehler während des Erstellungsprozesses angezeigt.

Öffne zum Starten des Programms das Kontextmenü des Projekts und wähle **Run As→ Local C/C++ Application** . Danach kannst du zum Starten den grünen Run-Knopf benutzen.



Aufgabe 1.6 Projekt debuggen

Du kannst dein Programm auch im Debug-Modus laufen lassen, um dir Schritt für Schritt den Ablauf anzusehen und mögliche Fehler zu finden. Zum Debuggen wählst du (wie auch bei Java) den Debug-Knopf („Käfer“) und Eclipse wird automatisch in die Debug-Perspektive wechseln.



Übung zum C/C++-Praktikum - Tag 1

Aufgabe 1.7 Erklärung des Hello-World Projekts

Wenn du *Hello World C++ Project* als Projekttyp auswählst, wird der Wizard automatisch ein Beispiel-Projekt erzeugen, welches **!!!Hello World!!!** auf der Konsole ausgibt. Als Einstieg gehen wir im Folgenden den generierten Code durch.

```
#include <iostream>
```

Die erste Zeile bindet den Header *iostream* ein. Dieser enthält unter anderem Klassen und Funktionen zur Ein- und Ausgabe. Beachte, dass es zwei verschiedene Möglichkeiten gibt, eine Header-Datei einzubinden - per **#include <...>** sowie per **#include "..."**. 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.

```
using namespace std;
```

Die zweite Zeile bewirkt, dass der **Namespace *std* eingebunden** wird. Ähnlich wie Package-Namen in Java dienen Namespaces (Namensräume) in C++ zur Strukturierung des Codes und zur Gruppierung von Elementen wie Funktionen und Klassen. Durch Namespaces können Namenskonflikte innerhalb von verschiedenen Bibliotheken vermieden werden. Anders als in Java sind Namespaces in C++ eher kurz, nur sehr selten verschachtelt und nicht an die Ordnerstruktur gebunden.

Der Namespace der C++Standardbibliothek *Standard Template Library (STL)* ist *std*. Um ein Element aus der *STL* zu verwenden, müsste man vor jedem Aufruf ein *std::* setzen, z.B. *std::cout* um Ausgaben auf der Konsole zu tätigen. Indem wir mittels **using namespace std;** den gesamten Namensraum einbinden, ist dies nicht mehr nötig. Man kann dies gut mit statischen Imports bei Java vergleichen. Bei Bedarf können auch einzelne Elemente des Namespaces eingebunden werden, z.B. **using std::cout;**.

```
int main() {
   ...
}
```

Die **Main-Funktion** stellt den Einstiegspunkt des Programms dar. Anders als in Java können Funktionen auch außerhalb von Klassen definiert und verwendet werden. Der zurückgegebene *int* signalisiert dem Aufrufer (Betriebssystem, Shell,...) den Erfolg oder Misserfolg der Ausführung. Typischerweise bedeutet **0 Erfolg** und Werte **kleiner 0 einen Fehler**.

Jedes vollständige C++ Programm muss **genau eine** *main()*-Funktion besitzen. Andernfalls wird der Linker mit der Fehlermeldung *undefined reference to 'main'* abbrechen.

```
cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
```

Durch **cout << ... << ...** können Werte auf der Konsole ausgegeben werden[1]. Ein **endl** beendet die aktuelle Zeile mit einem Zeilenvorschub. // leitet einen einzeiligen Kommentar ein. Mehrzeilige Kommentare werden in /* ... */ eingeschlossen.

Aufgabe 1.8 Häufige Compiler-Fehlermeldungen des gcc

```
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:

[1] Für weitere Informationen zur Kommandozeilenausgabe siehe http://www.cplusplus.com/doc/tutorial/basic_io/ und http://www.cplusplus.com/reference/iomanip/.

```
#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.

Aufgabe 1.9 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 32-bit machine
unsigned int ui; // unsigned int, 0 to 4294967295 on 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:

```
int n = 5;
// loop which runs until n is 0
while(n--) { // post-decrement n, i.e. decrement n but return previous value
   cout << n << endl;
}

// output: 4 3 2 1 0
```

Aufgabe 2 C++ Grundlagen, Funktionen und Strukturierung

Nach dieser kurzen Einführung darfst du dir jetzt selber die Hände schmutzig machen.

Wir lernen als zunächst die Kontrollstrukturen (if/for/while) von C++ kennen, die weitgehend gleich sind wie in Java. In den folgenden Aufgaben werden die Unterschiede und Gemeinsamkeiten stückweise erläutert.

Übung zum C/C++-Praktikum - Tag 1

a) Lege ein neues „Hello World“-Projekt an (*File → New → C++ Project → Hello World C++ Project*). Kompiliere und starte das Programm.

Jetzt haben wir einen Ausgangspunkt für eine eigene Implementierung.

b) Schrei eine Funktion *printStars(int n)*[2], die *n*-mal ein * auf der Konsole ausgibt und mit einem Zeilenvorschub 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.

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

c) Lagere dein Programm aus Aufgabenteil b) in eine eigene Datei aus. Gehe dazu folgendermaßen vor:

Erstelle eine neue Header-Datei *functions.h* und eine neue Sourcecode-Datei *functions.cpp*.

Binde *functions.h* in beide Sourcecode-Dateien mittels einer **#include** ein. Vergiss nicht, auch in *functions.cpp* den *std*-Namespace sowie *iostream* einzubinden, falls du dort Elemente der Standardbibliothek benutzen möchtest.

Vermeide bei größeren Programmen bitte unbedingt, den Namespace bereits im Header einzubinden! Dadurch würden alle Dateien, die den Header einbinden, den Namespace ebenfalls einbinden, was zu Namenskonflikten und anderen hässlichen Überraschungen führen kann. Benutze daher im Header immer voll qualifizierte Bezeichner (z.B. *std::string*, *std::ostream*).

Schreibe nun in *functions.h* **Funktionsprototypen** für die beiden Funktionen aus b). 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);*

Verschiebe deine beiden Funktionen nach *functions.cpp*. Fertig – die Ausgabe des Programms sollte sich nicht verändert haben.

d) Erweitere das in Aufgabenteil c) erstellte 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. Zum Einlesen verwendest du `cin >> variable`. Erstelle auch für diesen Aufgabenteil eine eigene Funktion und lagere diese nach *functions.cpp* aus.

e) 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
```

[2] 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.

```
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 *static*-Variablen erreicht werden. *static*-Variablen sind Variablen, die ihren alten Wert beim Wiedereintritt in die Funktion einnehmen. 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.

Aufgabe 3 Klassen

In der vorherigen Übung haben wir den Modifier **static** verwendet, um eine Funktion zu erstellen, welche fortlaufende Zeichen generiert. Diese Methode hat jedoch mehrere Nachteile, unter anderem kann man die Funktion nicht in mehreren Threads gleichzeitig nutzen. Deshalb werden wir nun eine Klasse schreiben, die das aktuelle Zeichen als Attribut enthält und durch Methoden ausgelesen und inkrementiert werden kann.

a) Lege ein neues Projekt an und füge dem Projekt eine neue leere Klasse *CharGenerator* hinzu. Obwohl Eclipse Klassenrümpfe automatisch generieren kann, wollen wir aus Übungsgründen die Klasse diesmal manuell erstellen. Erzeuge dazu einen Header *CharGenerator.h* und eine Sourcecode-Datei *CharGenerator.cpp*. Binde *CharGenerator.h* in *CharGenerator.cpp* und in *main.cpp* ein.

Erstelle nun den Klassenrumpf von *CharGenerator* in der Header-Datei. Die allgemeine Syntax lautet hierbei

```
class MyClass {
}; // semi-colon!
```

Genauso wie bei Funktionen wird auch bei Methoden von Klassen zwischen Deklaration und Implementierung unterschieden. Die Struktur der Klasse mit allen Attributen und Methodenprototypen wird im Header beschrieben, während die Sourcecode-Datei nur Implementierungen enthält. Da wir noch keine Methoden definiert haben, ist unsere Sourcecode-Datei entsprechend leer.

b) Füge Ihrer Klasse nun das *private* Attribut *char nextChar* hinzu, in dem das als nächstes auszugebende Zeichen gespeichert wird. Im Gegensatz zu Java werden in C++ die Access-Modifier *public/private/protected* nicht bei jeder Methode einzeln sondern blockweise angegeben. Dazu wird der jeweilige Access-Modifier an die gewünschte Stelle in der Klassendeklaration geschrieben und mit einem Doppelpunkt abgeschlossen. Alle darauffolgenden Deklarationen werden nun unter diesem Modifier erstellt. Beispiel:

```
class Foo {
public:
   void iAmPublic(); // public method
   // more public members
protected:
   void iAmProtected(); // protected method
   // more protected members
private:
   void iAmPrivate(); // private method
   ...
};
```

Füge das Attribut *nextChar* als **private** hinzu.

c) Noch wurde *nextChar* keinen Wert zugewiesen. Das wollen wir nun nachholen und einen Konstruktor für die Klasse *CharGenerator* erstellen, der *nextChar* auf einen Anfangswert setzt. Der Konstruktor wird als eine Methode ohne

Rückgabetyp deklariert, die den gleichen Namen wie die Klasse hat, und beliebige Parameter beinhalten kann, in unserem Fall zunächst

```
CharGenerator();
```

Erstelle den Konstruktorprototypen im *public*-Bereich der Klasse und eine Implementierung in der Sourcecode-Datei. Damit der Compiler weiß, zu welcher Klasse eine Methode/Konstruktor gehört, muss man vor dem Methodennamen den Scope der Klasse angeben.

```
CharGenerator::CharGenerator() { // Konstruktor von CharGenerator.
   ...
}
```

Um *nextChar* einen Wert zuzuweisen, werden wir eine sogenannte Initialisierungsliste verwenden. Diese beschreibt, wie die Attribute einer Klasse initialisiert werden. Dadurch wird garantiert, dass beim Eintritt in den Konstruktorrumpf alle Objektattribute bereits initialisiert sind und verwendet werden dürfen.

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 Attributennamen und ihren Initialisierungsargumenten in Klammern.

Die Reihenfolge der Initialisierungsliste sollte der Deklarationsreihenfolge entsprechen. Konstanten *müssen* in der Initialisierungsliste zugewiesen werden.

Beispiel:

```
CharGenerator::CharGenerator(): nextChar('a') {
}
```

Initialisiere nun *nextChar* mit 'a' wie beschrieben.

d) Schreibe nun eine *public* Methode *char generateNextChar()* die analog zur vorherigen Aufgabe das nächste auszugebende Zeichen zurückgibt. Teile die Methode in Prototyp und Implementierung auf. Vergiss auch hier nicht, den Scope der Klasse bei der Implementierung anzugeben:

```
char CharGenerator::generateNextChar() {
   ...
}
```

e) Teste deine Implementierung. Lege in der *main*-Funktion ein *CharGenerator*-Objekt mittels **CharGenerator charGen;** an. Ein **new** ist dabei nicht erforderlich (näheres dazu in der nächsten Vorlesung).

Rufe *generateNextChar()* einige Male auf und überprüfe das Ergebnis über die Konsole oder den Debugger.

f) Wir wollen nun angeben können, mit welchem Zeichen unser *CharGenerator* beginnen soll. Erweitere dazu den Konstruktor um einen Parameter *char initialChar* und ändere die Initialisierung von *nextChar*, damit dieser mit dem übergebenen Parameter gestartet wird.

Damit man nicht immer das Startzeichen angeben muss, kann man sogenannte *Default Parameter* angeben. Weise hierzu den Parameter im Prototypen einfach einen Wert zu. An der Implementierung muss nichts geändert werden.

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

Nun ist die Angabe des Startzeichens optional und kann auf Wunsch weggelassen werden. Bei der Definition eines Default-Parameters müssen für alle nachfolgenden Parameter ebenfalls Default-Werte angegeben werden, um Mehrdeutigkeiten beim Aufruf zu vermeiden.

Teste deine Implementierung sowohl mit als auch ohne Angabe des Startzeichens. Um ein Startzeichen anzugeben, lege das Objekt wie folgt an:

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

g) Erstelle eine neue Klasse *PatternPrinter* und füge ein *CharGenerator*-Objekt *charGenerator* als privates Attribut hinzu. Erstelle auch einen leeren, parameterlosen Konstruktor für *PatternPrinter*.

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.

h) Erstelle die Methoden *void printNChars(int n)*, *int readWidth()* und *void printPattern()*, die entsprechend *n* Zeichen auf die Konsole ausgeben, die Breite einlesen und unter Verwendungen aller Methoden das Muster aus der vorherigen Aufgabe auf der Konsole ausgeben. *printNChars* sollte hierbei *charGenerator* benutzen, um das nächste Zeichen ermitteln.

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

Aufgabe 4 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 *cout* der Klasse *ostream* kennengelernt, welche den <<-Operator überlädt, um Ausgaben von *std::string*, *int*,... komfortabel zu tätigen. In dieser Aufgabe wollen wir eine eigene Vektor-Klasse schreiben und einige Operatoren überladen.

Wir werden am Tag 4 auf dieser Aufgabe aufbauen und den Vektor um weitere Funktionen erweitern. Wirf deine Lösung also bitte nicht direkt weg. Falls du mit dieser Aufgabe bis dahin nicht fertig sein solltest, kannst du natürlich auf die Musterlösung zurückgreifen.

a) **Konstruktor** Schreibe eine Klasse *Vector3* mit den Attributen *a*, *b* und *c* vom Typ *double*, die die einzelnen Komponenten eines 3-dimensionalen Vektors darstellen. Die Klasse soll drei Konstruktoren besitzen, einen parameterlosen Default-Konstruktor, der den Vektor mit **0** initialisiert, einen Konstruktor mit 3 Parametern, der die einzelnen Vektor-Komponenten auf die gegebenen Werte setzt, und einen Copy-Konstruktor. Der Copy-Konstruktor wird aufgerufen, wenn das Objekt kopiert werden soll, z.B. für eine Call-by-Value Parameterübergabe. Er nimmt eine Objekt vom gleichen Typ wie die Klasse selbst (in unserem Fall *Vector3*) als Parameter und kopiert alle Attribute. Die Übergabe sollte dabei per const Reference geschehen. Näheres dazu in der Vorlesung. Bis dahin übernimm einfach die folgende Syntax für den Prototypen:

```
Vector3(const Vector3& other);
```

Dadurch kannst du *other* wie ein gewöhnliches *Vector3*-Objekt verwenden, es jedoch nicht verändern.

Füge jeden Konstruktor eine Ausgabe auf die Konsole hinzu, um beim Programmlauf den Lebenszyklus der Objekte nachvollziehen zu können

b) **Destruktor**

Erstelle außerdem einen Destruktor. Ein Destruktor ist eine Methode, die automatisch beim Löschen des Objektes aufgerufen wird. Die Syntax des Prototypen lautet

```
~Vector3();
```

und die Implementierung entsprechend

```
Vector3::~Vector3() {
   ...
}
```

Füge den Destruktor eine Ausgabe auf die Konsole hinzu, um beim Programmlauf den Lebenszyklus der Objekte nachvollziehen zu können

c) **Vektoraddition** Überlade den Operator + der Klasse *Vector3*, der eine komponentenweise Vektoraddition durchführt. Die Signatur lautet

```
Vector3 operator+(Vector3 rhs);
```

Das *Vector3* links ist dabei der Rückgabetyp der Überladung, der Parameter rhs die rechte Seite („right-hand-side“) des +-Operators. Dadurch, dass du den Operator als Member der Klasse deklarierst, nimmt die aktuelle Instanz hierbei automatisch die linke Seite der Operation an. Platziere den Prototyp im *public*-Bereich der *Vector3* Klasse.

Die Implementierung lautet dementsprechend:

```
Vector3 Vector3::operator+(Vector3 rhs) {
   ...
}
```

Du kannst nun innerhalb der Methode durch *a*, *b* und *c* auf eigene Attribute und über *rhs.a*, *rhs.b* und *rhs.c* auf Attribute der rechten Seite zugreifen.

d) **Vektorsubtraktion und Skalarprodukt**
Überlade auf die gleiche Weise auch die Operatoren — für eine komponentenweise Vektorsubtraktion sowie * für ein Skalarprodukt. Der Rückgabetyp eines Skalarprodukts ist kein *Vector3* sondern ein Skalar (*double*)!

Jetzt kannst du Vektoren durch *v1 + v2*, *v1 - v2* und *v1 * v2* addieren/subtrahieren und das Skalarprodukt bilden.

e) **Ausgabeoperator (*operator«*)** Um die Korrektheit unserer Implementierung zu testen, brauchen wie eine Ausgabemöglichkeit eines *Vektor3*-Objektes, idealerweise mit der gewohnten *cout* << ... Syntax. Dazu überladen wir den << Operator.

Diesmal müssen wir die Überladung **außerhalb** der *Vektor3*-Klasse definieren, weil auf unser *Vektor3*-Objekt nun auf der rechten Seite der Operation steht. Der Funktionsprototyp lautet

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

Als linke Seite wird hierbei ein *ostream*-Objekt (wie z.B. *cout*) erwartet. Um Ausgabeketten *cout* << ... << ... zu ermöglichen, muss das Ausgabeobjekt auch zurückgegeben werden. Damit das *ostream*-Objekt nicht jedes Mal kopiert wird, wird es als Referenz **&** durchgereicht.

Die Implementierung hat folgende Form:

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

Fülle die Funktion aus und kompiliere das Projekt. Vergiss nicht, den *iostream* Header und *std* Namespace einzubinden.

f) **Komponenten-Getter** Der Compiler wird mit der Fehlermeldung

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

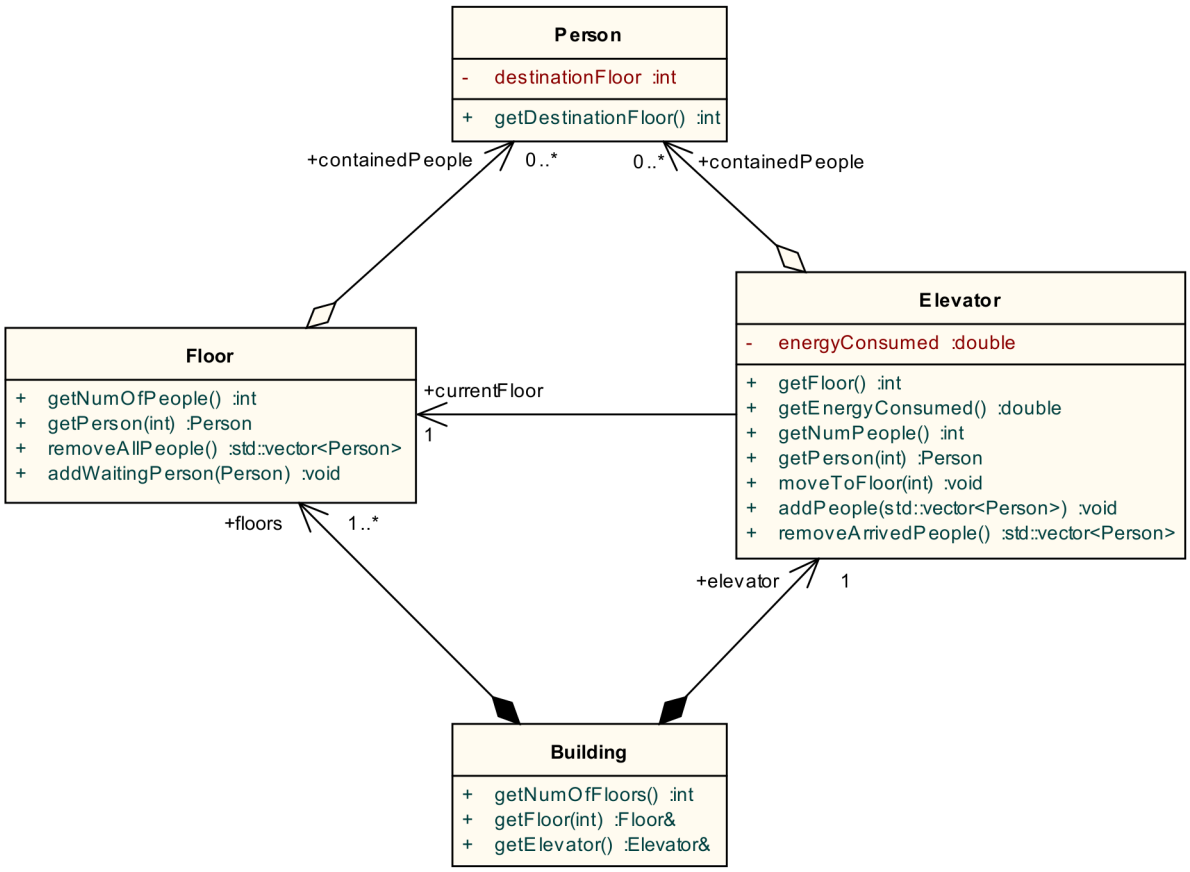
abbrechen. Dies liegt daran, dass die Attribute *a*, *b* und *c* privat und nur innerhalb der Klasse sichtbar sind. Erstelle deshalb Getter-Methoden für die einzelnen Vektorkomponenten und verwende diese in der Ausgabefunktion. Wenn alles funktioniert hat, kannst du beliebige *Vector3*-Objekte ausgeben:

```
cout << v << endl;
```

g) **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, sehen wir im nächsten Teil des Praktikums.

Aufgabe 5 Aufzug

In dieser Aufgabe soll ein Grundgerüst für den in der Vorlesung vorgestellten Aufzug-Simulator geschaffen werden. Folgendes Klassendiagramm veranschaulicht die Klassenstruktur.



a) **Klasse Person** Schreibe eine Klasse *Person*. Diese soll ein Attribut *int destinationFloor* haben, welches das Zielstockwerk angibt. Initialisiere das Attribut im Konstruktor und schreibe einen entsprechenden Getter. Füge dem Konstruktor außerdem eine Ausgabe auf die Konsole hinzu, um später beim Programmlauf den Erzeugungsprozess besser nachvollziehen zu können. Erstelle zudem einen Destruktor für *Person* sowie den Copy-Konstruktor. Vergiss nicht, im Copy-Konstruktor *destinationFloor* des anderes Objekts zu übernehmen. Erstelle auch hier eine Konsolenausgabe, um den Lebenszyklus darzustellen.

b) **Klasse Elevator** Schreibe eine Klasse *Elevator* mit den Attributen *int currentFloor*, *double energyConsumed* sowie *std::vector<Person> containedPeople*. Dabei soll *int currentFloor* die Nummer des aktuellen Stockwerks repräsentieren, *double energyConsumed* die verbrauchte Energie. In *containedPeople* werden die sich aktuell im Aufzug

befindlichen Menschen gespeichert. Dafür benutzen wir die Klasse *vector* aus dem *std* Namespace. Binde dazu den Header *vector* ein. Der Container *vector* kapselt ein Array und stellt eine ähnliche Funktionalität wie Javas *Vector* Klasse bereitt. 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.

Schreibe Getter für *currentFloor* und *energyConsumed*. Implementiere außerdem die Methoden

```
/** Returns number of people in Elevator */
int getNumPeople();

/** Returns i-th Person in Elevator */
Person getPerson(int i);
```

Du kannst mit *containedPeople.size()* auf die Länge eines *vector*s zugreifen und mittels *containedPeople.at(i)* auf das i-te Element.

Implementiere auch die Methode

```
/** Moves the elevator to given floor */
void moveToFloor(int floor);
```

die den Aufzug zu einem bestimmten Stockwerk bewegt. Passe die verbrauchte Energie sinnvoll an; addiere beispielsweise die Differenz zwischen dem aktuellen und dem Zielstockwerk hinzu.

Als letztes müssen wir die Methoden zum Ein- und Aussteigen implementieren:

```
/** Adds people to Elevator */
void addPeople(std::vector<Person> people);

/** Removes people which arrived at their destination */
std::vector<Person> removeArrivedPeople();
```

Du kannst dabei *containedPeople.push_back(Person)* nutzen, um eine einzelne Person zur Menge der Insassen hinzuzufügen. Um die Leute aussteigen zu lassen, die an ihrem Zielstockwerk angekommen sind, erstelle in der Methode zwei temporäre *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*.

c) **Klasse Floor** Schreibe die Klasse *Floor* mit dem Attribut *std::vector<Person> containedPeople*. Implementiere die folgenden Methoden:

```
/** Returns number of people on this floor */
int getNumPeople();

/** Returns i-th Person on this floor */
Person getPerson(int i);

/** Adds a Person to this floor */
void addWaitingPerson(Person h);

/** Removes all persons from this floor and return them */
std::vector<Person> removeAllPeople();
```

Nutze *containedPeople.clear()* um alle Elemente eines *vector*s zu löschen.

d) **Klasse Building** Schreibe eine Klasse *Building*. Der Konstruktor soll dabei die Anzahl der Stockwerke als Parameter erhalten. Füge einen Aufzug *elevator* sowie *std::vector<Floor> floors* als private Attribute hinzu. Implementiere einen Getter, der den Aufzug als Referenz (*Elevator&*) zurückgibt. Implementiere dazu die folgenden Methoden:

```
/** Returns number of floors */
int getNumOfFloors();

/** Returns a certain floor as reference*/
Floor& getFloor(int floor);
```

e) Um die Benutzung des Simulators von außen zu vereinfachen und lange Aufrufketten wie

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

zu vermeiden, werden wir *Building* einige weitere Methoden hinzufügen. Der Simulator sollte nur mit Methoden der Klasse *Building* kommunizieren. Implementiere hierfür folgende Komfortmethoden:

```
/**
 * Lets people on current floor go into the elevator.
 */
void letPeopleIn();

/** Removes people from elevator on current floor which arrived at their destination */
std::vector<Person> removeArrivedPeople();

/** Moves the building's elevator to given floor */
void moveElevatorToFloor(int i);

/** Adds a person to given floor */
void addWaitingPerson(int floor, Person p);
```

f) Teste deine Implementierung. 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.