Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 1. Tag

Aufgabe 1 Eclipse CDT

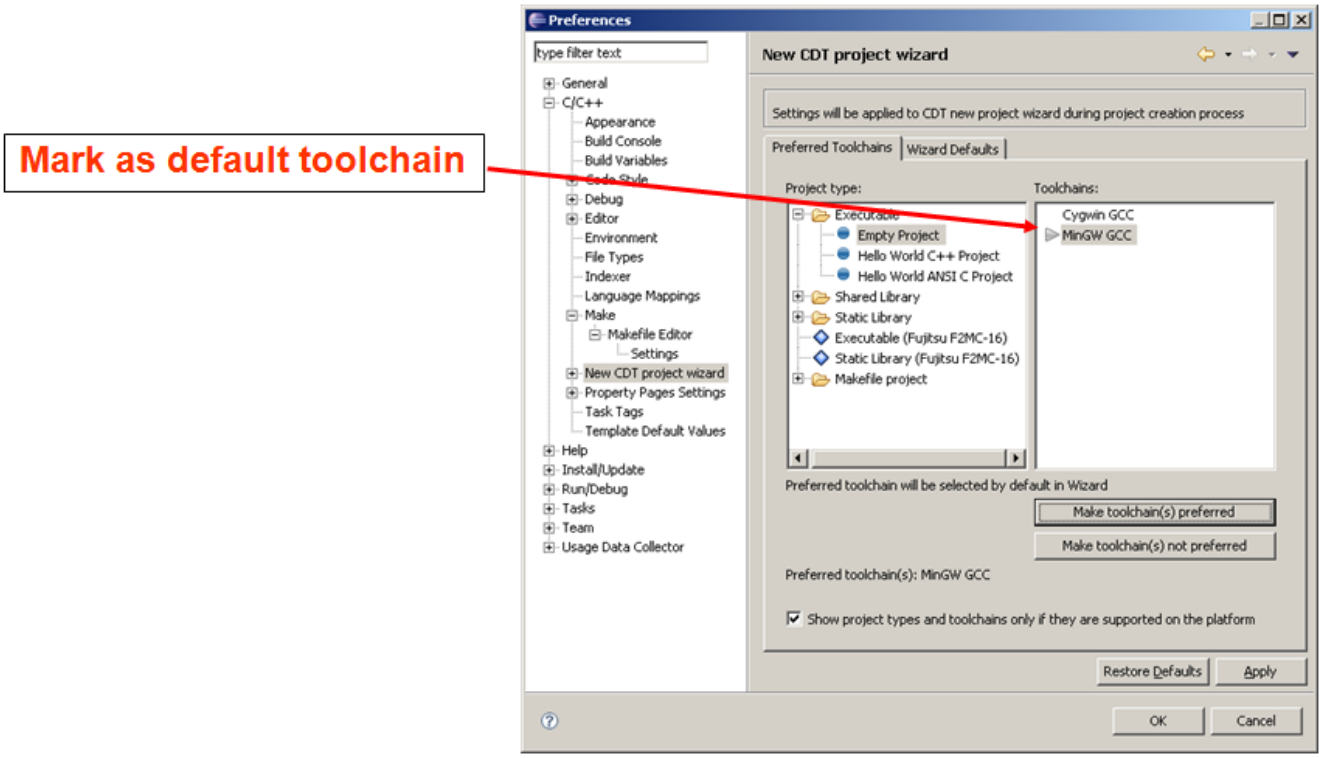
Für alle Übungen des C/C++ Praktikums werden wir Eclipse zusammen mit dem C/C++ Development Tooling (CDT) und dem MinGW gcc Compiler verwenden. Es wird davon ausgegangen, dass Sie den generellen Umgang Eclipse bereits aus der Java-Programmierung kennen und nur die Unterschiede zur C++-Programmierung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

Alle Tools sind bereits auf den Poolrechnern vorinstalliert. Falls Sie mit ihrem Nodebook arbeiten möchten, können Sie den Installer von http://130.83.199.65/download/cplusplus/Eclipse-CPPP.exe beziehen.

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

Aufgabe 1.1 Toolchain einstellen

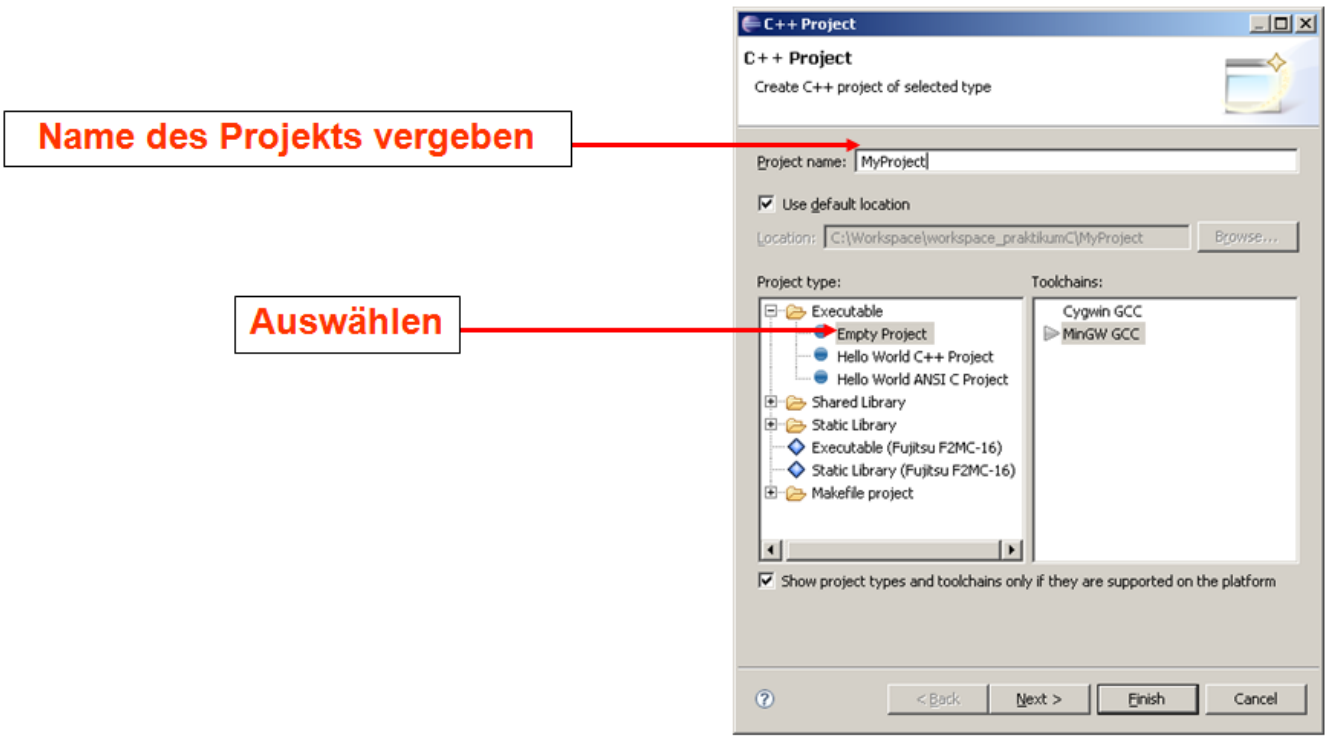
Als erstes muss MinGW als die bevorzugte C/C++ Toolchain eingestellt werden. Öffnen Sie dazu den Menüpunkt **Window → Preferences** und wählen Sie **C/C++ → New CDT project wizard**. Markieren Sie nun **MinGW GCC** rechts in der Liste und klicken Sie auf **Make toolchain(s) preferred**. Schließen Sie den Dialog anschließend mit **OK**.



Aufgabe 1.2 Neues Projekt anlegen

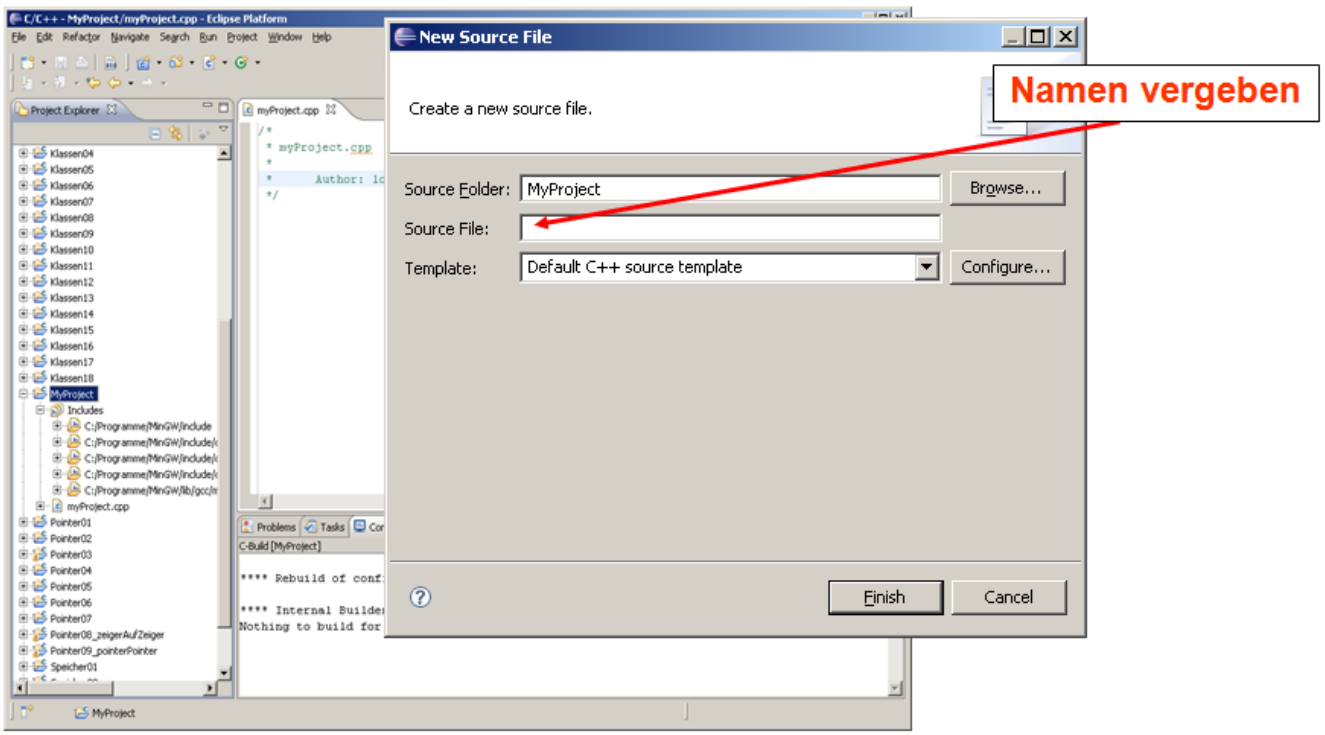
Um ein neues Projekt anzulegen, wählen Sie **File → New → C++ Project** im Eclipse Menü. Geben Sie den gewünschten Projektnamen ein und wählen Sie **Empty Project** bzw. **Hello World C++ Project** als Projekttyp aus. Die Toolchain sollte

bereits automatisch auf MinGW voreingestellt sein.



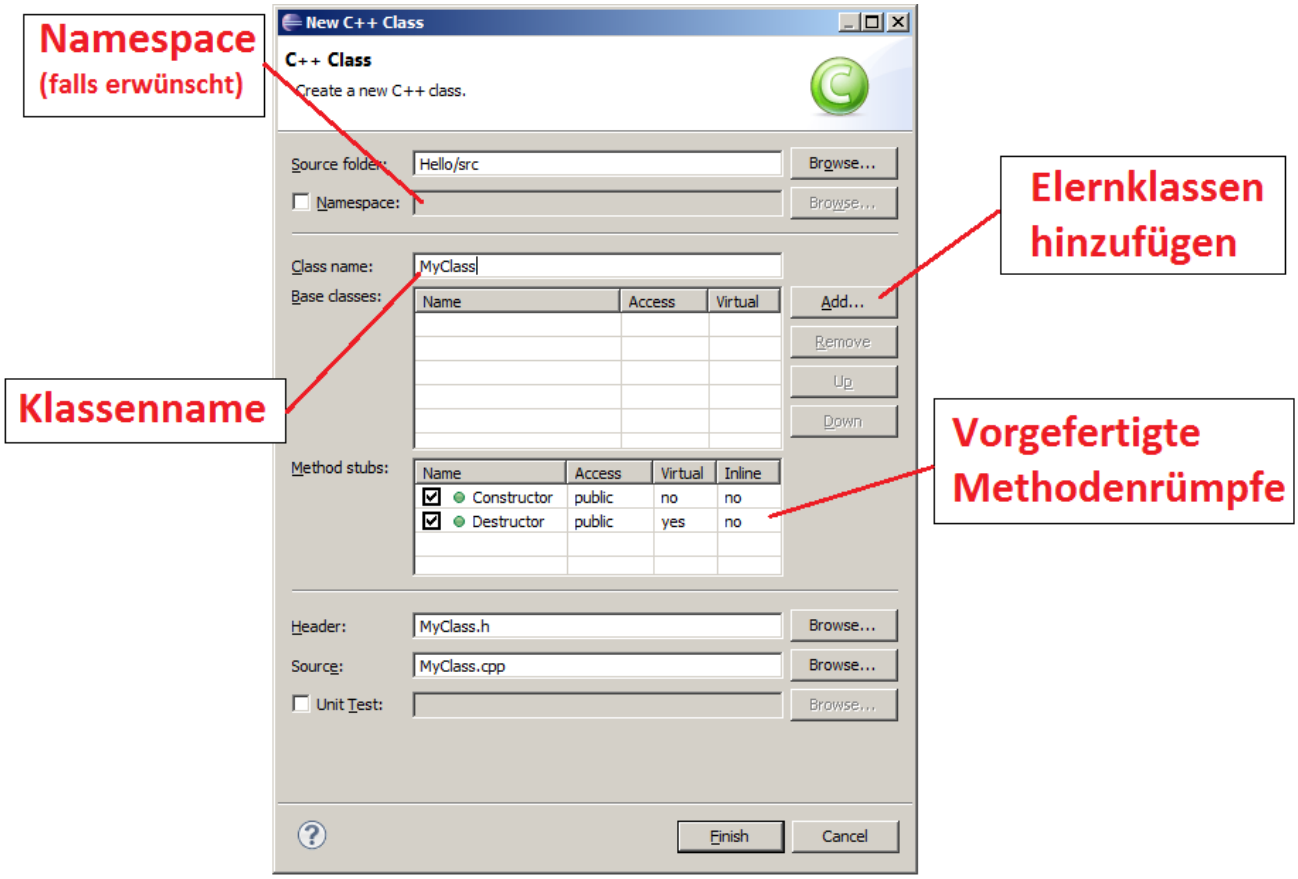
Aufgabe 1.3 Neue Dateien hinzufügen

Um eine neue Sourcecode-Datei zum Projekt hinzuzufügen, klicken Sie mit der rechten Maustaste auf das Projekt und wählen Sie **New → Source File**. Geben Sie einen Dateinahmen ein und bestätigen Sie mit **Finish**. Verfahren Sie analog, um Header-Dateien zu erstellen, wählen Sie jedoch **New → Header File** im Kontextmenü. Sourcecode-Dateien tragen in der Regel die Endung *.cpp*, Header-Dateien *.h*, *.hpp* oder gar keine Endung. Wir empfehlen Ihnen jedoch *.h* zu benutzen.



Aufgabe 1.4 Neue Klassen hinzufügen

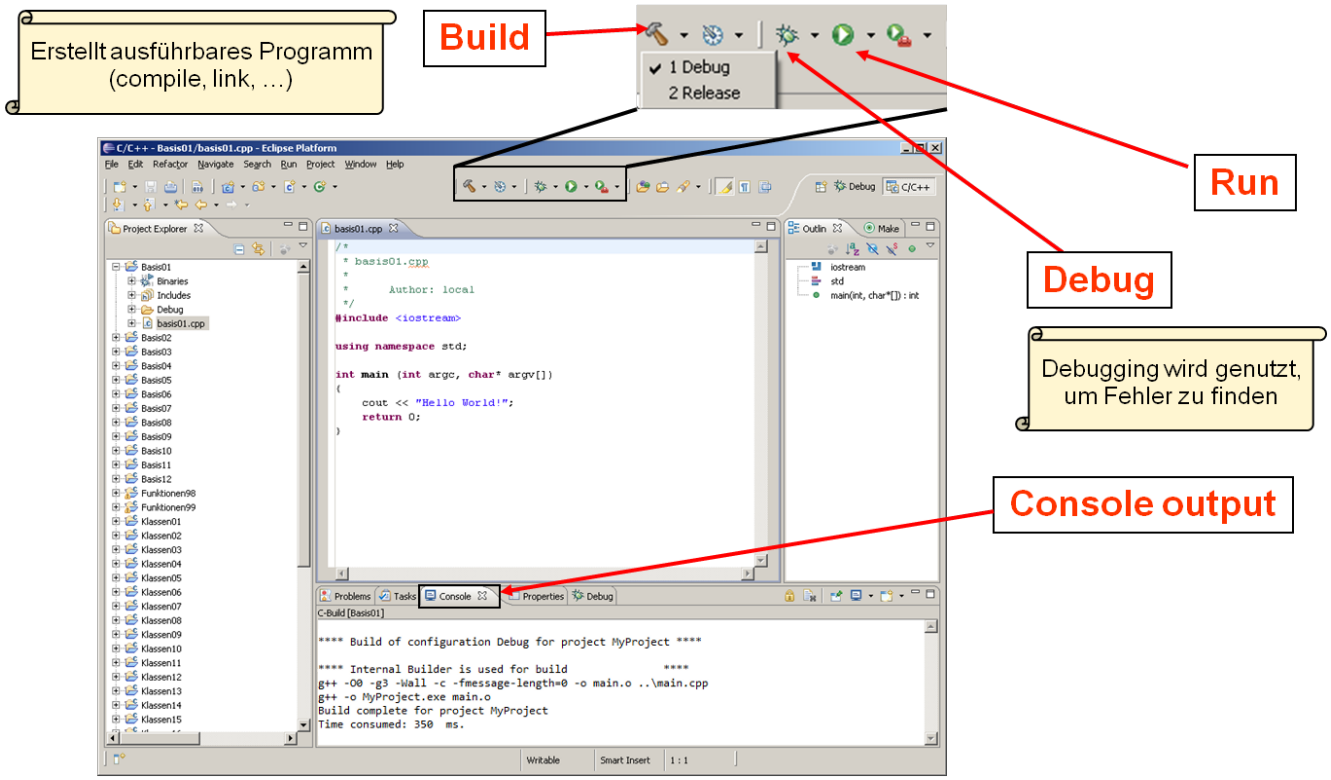
Für eine Klasse kann man sowohl den Header als auch die Sourcecode-Datei in einem Schritt erzeugen. Wählen Sie dazu **New → Class** im Kontext-Menü des Projekts, um den Wizard zu starten. Geben Sie den Namen und bei Bedarf den Namespace sowie weitere Informationen wie z.B. die Elternklassen ein (dazu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



Aufgabe 1.5 Projekt kompilieren und starten

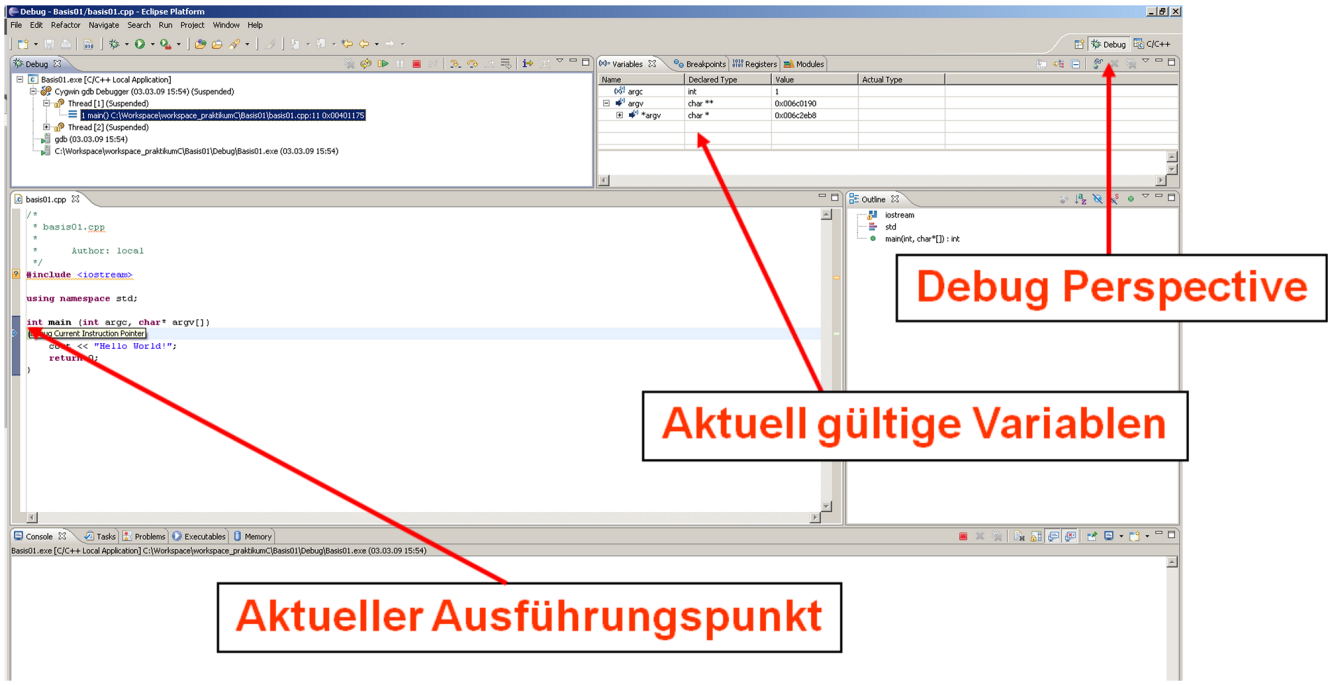
Bevor ein C++-Programm gestartet werden kann, muss es compiliert werden. Im Gegensatz zu Java wird der Compiler nicht automatisch von Eclipse im Hintergrund aufgerufen, sondern muss manuell gestartet werden. Um das aktuell offene Projekt zu kompilieren, klicken Sie auf das **Build**-Symbol in der Eclipse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

Um ein kompiliertes Projekt zum ersten mal zu starten, öffnen Sie mit der rechten Maustaste das Kontextmenü des Projekts und wählen Sie **Run As→ Local C/C++ Application** . Danach können Sie zum Starten den grünen Run-Knopf benutzen.



Aufgabe 1.6 Projekt debuggen

Falls sich im Projekt Fehler einschleichen, die nicht durch einfaches Hingucken behoben werden können, kann es sinnvoll sein, ein Projekt zu debuggen, also die Anweisungen Schritt für Schritt die durchzugehen und die aktuellen Variablen anzuschauen. Dazu muss das Projekt durch drücken des Debug-Knopfes im Debug-Modus gestartet werden. Zum Debuggen wird Eclipse automatisch in die Debug-Perspektive wechseln.



Übung zum C/C++-Praktikum - Tag 1

Aufgabe 1.7 Erklärung des Hello-World Projekts

Wenn Sie *Hello World C++ Project* als Projekttyp ausgewählt haben, wird der Wizard automatisch ein Beispiel-Projekt erzeugen, welches **!!!Hello World!!!** auf der Konsole ausgibt. Wir werden den generierten Code nun stückweise durchgehen und näher erklären.

```
#include <iostream>
```

Die erste Zeile bindet den Header *iostream* ein. Dieser enthält unter anderem Klassen und Funktionen zur Ein- und Ausgabe. Beachten Sie, 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 und nur sehr selten verschachtelt.

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::* setzten, z.B. *std::cout* um Ausgaben auf der Konsole zu tätigen. Durch die Einbindung des kompletten Namespaces mittels **using namespace std;** ist dies nicht mehr nötig. Bei Bedarf können auch einzelne Elemente des Namespaces eingebunden werden, z.B. durch **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. Durch die Rückgabe eines *int*s kann das Programm ihrem Aufrufer, also dem Betriebssystem oder der Shell, Erfolg oder Misserfolg der Ausführung mitteilen. Typischerweise signalisieren 0 Erfolg und Werte kleiner 0 einen Fehler. Auf das Programm selbst hat der Rückgabewert keinen Einfluss.

Jedes vollständige C++ Programm muss 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 verschiedene Werte auf der Konsole ausgegeben werden. 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. Beachten Sie, 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, diese aber nirgendwo tatsächlich definiert. Überprüfen Sie 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 größtenteils identisch zu den aus Java. Allerdings sind alle Ganzzahl-Typen in C++ sowohl mit als auch ohne Vorzeichen verfügbar. Standardmäßig sind Vorzeichen möglich, mittels **unsigned** kann man aber vorzeichenlose Variablen deklarieren. Dabei wird der Definitionsbereich entsprechend erweitert, da nun ein Bit mehr zur Verfügung steht.

```
int i; // signed int, -2147483648 to +2147483647 on 32-bit machine
unsigned int ui; // unsigned int, 0 to 4294967295 on 32-bit machine
```

Eine andere Besonderheit von C++ ist dass Ganzzahlwerte implizit in *bool*-Werte 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

Als erstes machen wir uns mit der grundlegenden C++-Syntax vertraut. Diese ist in Grundzügen Java sehr ähnlich, insbesondere die Klammerung und die Kontrollstrukturen wie if/for/while sind beinahe identisch. In den folgenden Aufgaben werden die Unterschiede und Gemeinsamkeiten stückweise erläutert. Falls nicht anders angegeben, können Sie davon ausgehen, dass Sie die gleiche Syntax wie in Java benutzen dürfen.

a) Legen Sie ein neues „Hello World"-Projekt an (*File → New → C++ Project → Hello World C++ Project*). Versuchen Sie, das Programm zu kompilieren und zu starten.

b) Schreiben Sie eine Funktion *print_star(int n)*, die *n*-mal ein * auf der Konsole ausgibt und mit einem Zeilenvorschub abschließt. Ein Aufruf von *print_star(5)* sollte also folgende Ausgabe generieren:

   *****

   Platzieren Sie die Funktion noch vor der *main*, da sie sonst nicht ohne weiteres aufgerufen werden kann. Benutzen Sie die erstellte Funktion *print_star(int n)*, um eine weitere Funktion zu schreiben, die eine Figur wie unten dargestellt ausgibt. Verwenden Sie hierzu Schleifen, kodieren sie die Ausgabe der einzelnen Zeilen nicht direkt!

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

c) Lagern Sie ihr Programm aus Aufgabenteil b) in eine eigene Datei aus. Gehen Sie dazu folgendermaßen vor:

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

Binden Sie *functions.h* in beide Sourcecode-Dateien mittels einer entsprechenden **#include**-Anweisung ein. Vergessen Sie nicht, auch in *functions.cpp* den *std*-Namespace sowie *iostream* einzubinden, falls Sie dort Elemente der Standardbibliothek benutzen möchten. Vermeiden Sie insbesondere bei echten 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 Überraschungen führen kann. Benutzen Sie deshalb den Scope-Operator **::** im Header-Code.

Schreiben Sie nun in *functions.h* Funktionsprototypen für die beiden Funktionen aus b). Funktionsprototypen dienen dazu, dem Compiler mitzuteilen, dass eine Funktion mit gegebenen Parametern existiert und aufgerufen werden kann, ohne die Implementierung bereitzustellen. Ein Prototyp ist im wesentlichen eine mit **;** abgeschlossene Signatur der Funktion ohne Funktionsrumpf. Der Prototyp von *print_star(int n)* lautet dementsprechend *void print_star(int n);*

Kopieren Sie Ihre beiden Funktionen nach *functions.cpp*. Nun können aus der main.cpp heraus Funktionen aus *functions.cpp* aufrufen werden.

d) Erweitern Sie 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. Gibt der Benutzer eine zu große Breite ein, wird er darüber informiert und erneut gefragt. Zum Einlesen benutzen Sie `cin >> variable`. Erstellen Sie auch für diesen Aufgabenteil eine eigene Funktion und lagern Sie diese nach *functions.cpp* aus.

e) Statt einem einzigen Zeichen soll nun das fortlaufende Alphabet ausgegeben werden. Sobald das Ende des Alphabets erreicht wurde, soll wieder bei *a* begonnen werden. Beispiel:

```
abc
de
f
gh
 ijk
```

Implementieren Sie dazu eine Funktion *char next_char()*. Diese soll bei jedem Aufruf das nächste auszugebende Zeichen von Typ *char* zurückgeben, beginnend bei '*a*'. Dazu muss sich *next_char()* 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, da es zur Interferenz bei der Verwendung der statischen Variable kommen könnte. Deshalb werden wir nun eine Klasse schreiben, welche das aktuelle Zeichen als Attribut enthält und durch Methoden ausgelesen und inkrementiert werden kann. Dadurch können mehrere Instanzen der Klasse nebeneinander existieren, ohne die Zeichenausgabe gegenseitig zu stören.

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

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

```
class <Name> { // Echter Name ohne <>
}; // Semikolon am Ende beachten!
```

   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, unsere Sourcecode-Datei entsprechend leer.

b) Fügen Sie Ihrer Klasse nun das 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 ichBinPublic(); // public Methode
    ... // weitere Methoden/Attribute
protected:
    void ichBinProtected(); // protected Methode
    ...
private:
    void ichBinPrivate(); // private Methode
    ...
};
```

   Fügen Sie 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 definiert, die den gleichen Namen wie die Klasse hat, und beliebige Parameter beinhalten kann, in unserem Fall also

```
CharGenerator();
```

   Erstellen Sie 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, welches Attribut mit welchem Wert initialisiert wird, bzw. welcher Konstruktor für die Erzeugung von Objektattributen verwendet werden soll. 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. Beispiel:

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

Initialisieren Sie nun *nextChar* mit 'a' durch die oben beschriebene Weise.

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

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

e) Testen Sie Ihre Implementierung. Legen Sie in der main ein *CharGenerator*-Objekt mittels **CharGenerator charGen;** an. Ein **new** ist dabei nicht erforderlich (näheres dazu in der nächsten Vorlesung).

Machen Sie einige Aufrufe von *generateNextChar()* und geben Sie das Ergebnis auf der Konsole aus.

f) Wir wollen nun angeben können, mit welchem Zeichen unser *CharGenerator* beginnen soll. Erweitern Sie dazu den Konstruktor um einen Parameter *char initialChar* und ändern sie 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. Weisen Sie hierzu den jeweiligen 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. Beachten Sie, dass bei der Definition eines Default-Parameters für alle nachfolgenden Parameter ebenfalls Default-Werte angegeben werden müssen, um Mehrdeutigkeiten beim Aufruf zu vermeiden.

Testen Sie Ihre Implementierung sowohl mit als auch ohne Angabe des Startzeichens. Um ein Startzeichen anzugeben, legen Sie das Objekt mittels **CharGenerator charGen('x');** an.

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

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) Erstellen Sie 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.

Testen Sie Ihre Implementierung, indem Sie ein *PatternPrinter*-Objekt anlegen und *printPattern()* darauf aufrufen.

Übung zum C/C++-Praktikum - Tag 1

Aufgabe 4 Operatorüberladung

In C++ besteht die Möglichkeit, Operatoren wie +, *, ... in Zusammenhang mit eigenen Klassen zu überladen. Das bedeutet man kann selbst spezifizieren, was beim Verknüpfen von Objekten mit einem Operator geschehen soll, um zum Beispiel den Quellcode übersichtlicher zu gestalten. Sie haben bereits das Objekt *cout* der Klasse *ostream* kennen gelernt, welche den <<-Operator überladen hat, um Ausgaben 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. Behalten Sie dies bitte im Hinterkopf und löschen Sie Ihre Lösung nicht. Falls Sie mit dieser Aufgabe bis dahin nicht fertig sein sollten, können Sie natürlich auch die Musterlösung als Basis nehmen.

a) Schreiben Sie eine Klasse *Vector3* mit den Attributen *a*, *b* und *c* vom Typ *double*, die die einzelnen Komponenten eines 3-dimensionalen Vektors darstellt. 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 den gegebenen Wert 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 erfahren Sie morgen. Bis dahin übernehmen Sie einfach die folgende Syntax für den Prototypen:

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

   Sie können *other* wie ein gewöhnliches *Vector3*-Objekt verwenden, es jedoch nicht verändern.

   Erstellen Sie 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() {
       ...
   }
   ```

   Fügen Sie in beide Konstruktoren und den Destruktor eine Ausgabe auf die Konsole hinzu, um beim Programmlauf den Lebenszyklus der Objekte besser nachvollziehen zu können

b) Überladen Sie 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. Das eigene Objekt nimmt hierbei automatisch die linke Seite der Operation an. Platzieren Sie den Prototyp im *public*-Bereich der *Vector3* Klasse.

   Die Implementierung lautet dementsprechend

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

   Sie können 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.

c) Überladen Sie auf die gleiche Weise auch die Operatoren − für eine komponentenweise Vektorsubtraktion sowie * für ein Skalarprodukt. Beachten Sie, dass der Rückgabetyp eines Skalarprodukts kein *Vector3* sondern ein Skalar (*double*) ist!

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

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

Diesmal müssen wir die Überladung **außerhalb** der *Vektor3*-Klasse definieren, weil auf unser *Vektor3*-Objekt nun auf der rechten Seite der Operation stehen muss. 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 **&** übergeben.

Zusammengefasst hat die Implementierung folgende Form:

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

Füllen Sie die Funktion aus und versuchen Sie, das Projekt zu kompilieren. Vergessen Sie nicht, den *iostream* Header und *std* Namespace einzubinden.

e) 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 griffbereit sind. Erstellen Sie deshalb Getter-Methoden für die einzelnen Vektorkomponenten und verwenden Sie diese in der Ausgabefunktion. Wenn alles funktioniert hat, können Sie beliebige *Vector3*-Objekte über

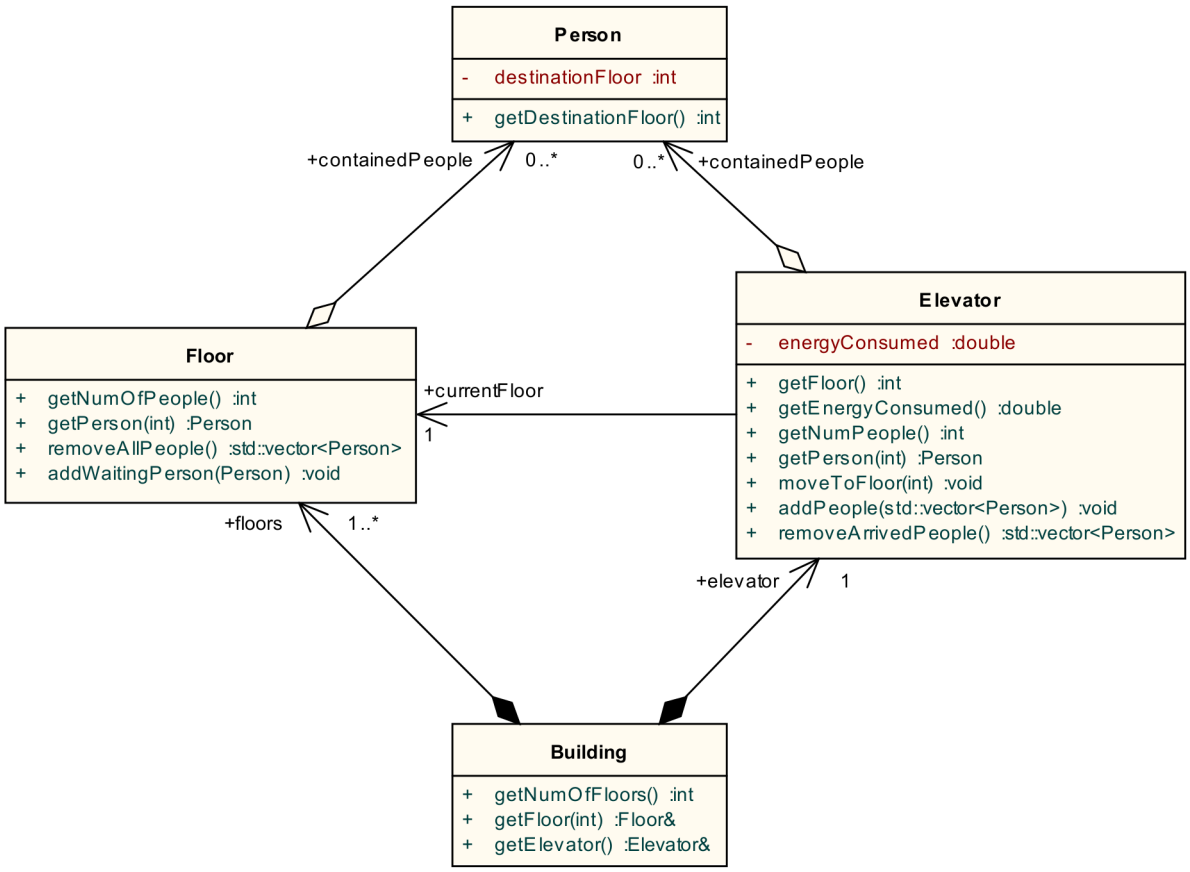
```
cout << v << endl;
```

ausgeben.

f) Testen Sie ihre bisher definierten Methoden und Funktionen. Probieren Sie auch Kombinationen von verschiedenen Operatoren aus und beobachten Sie das Ergebnis. Schreiben Sie auch eine einfache Funktion, die Vektoren als Parameter nimmt. Wie Sie sehen können, 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 es vermieden werden kann, erfahren Sie 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 sollte die Klassenstruktur veranschaulichen.



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

b) Schreiben Sie 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. Binden Sie dazu den Header *vector* ein. *vector* ist ein Container, welches ein Array kapselt und eine ähnliche Funktionalität wie Javas *Vector* Klasse bereitstellt. Das **<Person>** in *std::vector<Person>* ist ein Template-Parameter und besagt, dass in dem Container *Person*-Objekte gespeichert werden sollen.

Schreiben Sie Getter für *currentFloor* sowie *energyConsumed*. Implementieren Sie außerdem die Methoden

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

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

Sie können mit *containedPeople.size()* auf die Länge eines vectors zugreifen und mittels *containedPeople.at(i)* auf das i-te Element.

Implementieren Sie auch die Methode

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

die den Aufzug zu einem bestimmten Stockwerk fahren lässt. Passen Sie die verbrauchte Energie entsprechend an, addieren Sie z.B. die Differenz zwischen dem aktuellen und dem Zielstockwerk hinzu.

Als letztes müssen wir die Methoden zum Ein- und Aussteigen in/aus dem Aufzug schreiben.

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

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

Sie können 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, erstellen Sie in der Methode zwei temporäre *vector*-Container *stay* und *arrived*. Iterieren Sie nun über alle Leute in dem Aufzug und prüfen Sie, ob das Zielstockwerk der Person mit dem aktuellen Stockwerk des Aufzugs übereinstimmt. Wenn ja, lassen Sie die Person aussteigen, indem Sie sie zu der arrived-Liste mittels *push_back()* hinzufügen. Andernfalls muss die Person im Aufzug verbleiben (stay-Liste). Geben Sie am Ende die arrived-Liste zurück, und ersetzen Sie *containedPeople* durch *stay*.

c) Schreiben Sie die Klasse *Floor* mit dem Attribut *std::vector<Person> containedPeople*. Implementieren Sie die Methoden

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

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

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

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

Nutzen Sie *containedPeople.clear()* um alle Elemente eines Vectors zu löschen.

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

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

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

analog zur vorherigen Aufgabe.

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. Diese sollen als Zugriffspunkte für den Simulator dienen. Implementieren Sie hierfür folgende Methoden:

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

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

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

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

f) Testen Sie Ihre Implementierung. Erstellen Sie dazu zunächst ein Gebäude und fügen Sie 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
```

Implementieren Sie 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;
    Let all people on Floor into the Elevator;
    while Elevator has people do
        Move Elevator to destination Floor of first Person in Elevator;
        Remove arrived people;
    end
end
```

Geben Sie am Ende auch die benutze Energie aus. Schauen Sie sich die Ausgabe genau an und versuchen Sie nachzuvollziehen, warum Personen so oft kopiert werden. Denken Sie daran, dass diese bei einer Übergabe als Argument kopiert werden.