Skript

zum
Hochschulpraktikum

Industrielle Softwareentwicklung
für Ingenieur:innen / C++

Stand 25.09.2025

Lehrstuhl für Automatisierung und Informationssysteme

0 Inhaltsverzeichnis

1 Tage 1–3

1.1 Einleitung

Das vorliegende Skript stellt eine Einführung in die Programmierung mit C++ dar. Zum Verständnis des Skriptes werden grundlegende Kenntnisse in C vorausgesetzt.

Das Skript ist in 4 Kapiteln unterteilt: Kapitel 1 beinhaltet den Arbeitsinhalt der ersten drei Tage und behandelt die Grundlagen der Programmierung mit C++. Kapitel 2 und 3 sind für die Tage 4 bzw. 5 vorgesehen und begleitet Schritt für Schritt die Programmierung einer Robotersteuerung zunächst an einer Computersimulation. In Kapitel 4 wird abschließend die Programmierung des physischen Roboterfahrzeugs behandelt (Tag 6).

Zunächst werden in Kapitel 1 Abschnitt 1.4 bis 1.7 die Arbeitsumgebung, das Kompilieren von C++-Programmen und die verwendete Entwicklungsumgebung *Eclipse* vorgestellt. Abschnitt 1.8 bis 1.13 wiederholen dann einige aus C bekannte Konstrukte wie Kontrollstrukturen und Pointer. Außerdem werden auch für C++ spezifische Grundlagen wie Referenzen und die Verwendung des dynamischen Speichers erläutert.

Es folgt in Abschnitt 1.14 bis 1.16 eine Einführung in das Paradigma der Objektorientierung und seine Umsetzung in C++. Das Konzept einer Klasse und weitere damit verbundene Konzepte wie Kapselung, Instanziierung von Objekten, etc. werden ausführlich beschrieben. Danach wird in Abschnitt 1.17 die Objektorientierung um die Idee der Vererbung erweitert.

Abschnitt 1.18 bis 1.21 beschäftigen sich mit den fortgeschrittenen Programmiertechniken Rekursion, Dateioperationen, Ausnahmen und der Verwendung von Listen in C++. Den Abschluss bildet Abschnitt 1.22, der die Verwendung von externen Bibliotheken in C++ skizziert.

Alle Themengebiete werden von Aufgaben begleitet, deren erfolgreiche Bearbeitung die Studenten nachweisen müssen. Die Zusatzaufgabe in Abschnitt 1.18 ist davon ausgenommen und ist für Interessierte.

Das Kapitel 1 ist auf eine Bearbeitungszeit von drei Tagen ausgelegt. Es wird empfohlen, am ersten Bearbeitungstag mindestens Abschnitt 1.13 und am zweiten mindestens Abschnitt 1.17 abzuschließen.

1.2 Überblick

Tag	Aufgaben
1	Taschenrechner Maya Zahlen Verfahren von Heron Palindrome Buchstaben und Wörter Quadratische Funktionen Mathematische Funktionen / Die Quadratwurzel sqrt Speicher und Felder Bubblesort
2	Flaschen Datum – Finden Sie die Fehler Erweiterte Datumsklasse Fahrzeuge Quadrate Vererbung und Überschreibung Virtuelle Methoden Polymorphie
3	Rekursion Fibonacci-Folge Springerproblem (Zusatzaufgabe) Dateien lesen und schreiben Ausnahmen Liste von Koordinaten ncurses
4	Erstellen der Klasse KeyboardControl Erstellen der Communicate-Methode Einbinden der Schnittstelle und Erstellen der Step-Methode Ausführen des Programms Regler Tastatursteuerung mit Regelung
5	Positionsschätzung Koordinatenliste Kontrollmethoden Geschwindigkeitsberechnung und –übergabe Die Methode Communicate Die Methode Step
6	Cross-Compiling-Projekt Tastatursteuerung (Hardware) Manöversteuerung (Hardware) Kombinierte Tastatur- und Manöversteuerung (Hardware) (Zusatzaufgabe)

1.3 Motivation

War vor ein paar Jahren das Programmieren von Software vor allem Informatikern überlassen, so wird heute von annähernd jeder Ingenieurin und jedem Ingenieur erwartet, selbst Code zu verstehen und zu schreiben. Dabei wird jedoch oft unterschätzt, wie wichtig es ist nicht nur „theoretisch“ zu wissen wie man programmiert, sondern es auch notwendig ist, sich an den Computer zu setzten und ein Programm selbst zu schreiben. Neben dem eigentlichen Schreiben des Codes gehört jedoch auch die Planung und das Debuggen zu den Kernkompetenzen eines jeden Entwicklers. Dieses Praktikum bietet Ihnen die Möglichkeit, Ihre Programmierkenntnisse zu verbessern und einen Einblick in die praktische Entwicklung von C++ und das Handwerkszeug für komplexere Programme zu erhalten.

Auch wenn Programmieren zum größten Teil selbständiges Arbeiten ist, können verschiedene Quellen sehr gut zum Verständnis beitragen und bei der Syntax weiterhelfen. Informative Seiten hierfür können www.stackoverflow.com, www.willemer.de/informatik/cpp, http://de.cppreference.com oder auch einfach eine Suche bei Google sein.

1.4 VM-Ware Player starten

Kopieren Sie zunächst die Virtual Machine vom öffnetlichen Laufwerk in Ihr persönliches Laufwerk auf C: indem sie unter `DieserPC\C:\Benutzer\Öffentlich\VirtualMachines\SefiC++Praktikum_VM\Ubuntu_IDE_SEFI`

den Ordner „Ubuntu_IDE_SEFI“ nach `DieserPC\C:\Benutzer\"tum-kennung"`

kopieren.

Starten Sie im Anschluss den VMware Player. Binden Sie über die Option *Open a Virtual Machine* die Virtuelle Maschine Ubuntu_IDE_SEFI ini ihrem persönlichen Benutzer-Ordner ein.

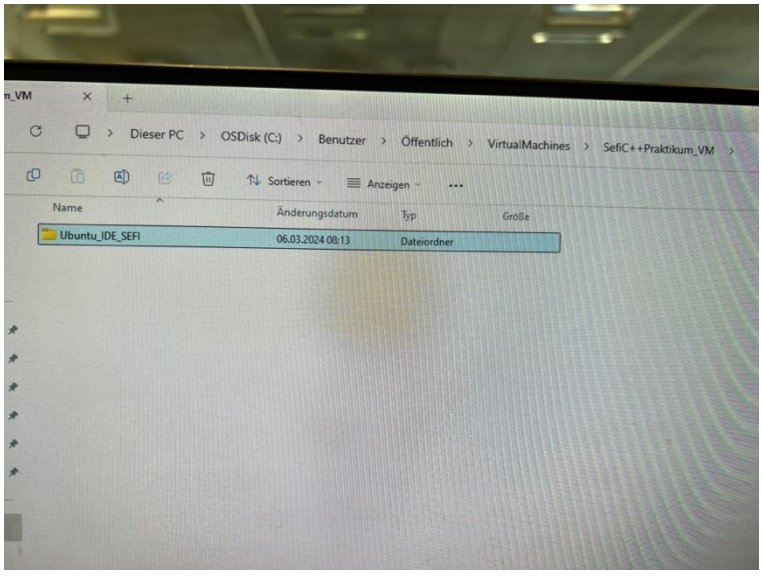


Abbildung 1: Pfad der Virtuellen Maschine

Wenn Sie in Präsenz mit einem Rechner der Universität am Praktikum teilnehmen, muss folgender Arbeitsschritt durchgeführt werden: Nach dem Einbinden der VM muss das Netzlaufwerk eingebunden werden. Wählen Sie hierzu die Ubuntu VM im VMware Player aus und klicken sie auf den Link Edit virtual machine settings (siehe Abbildung 2). Eine weitere Möglichkeit, falls die VM bereits gestartet worden ist, stellt der folgende Pfad dar: *Player → Manage → Virtual Machine Settings…* (siehe Abbildung 3).

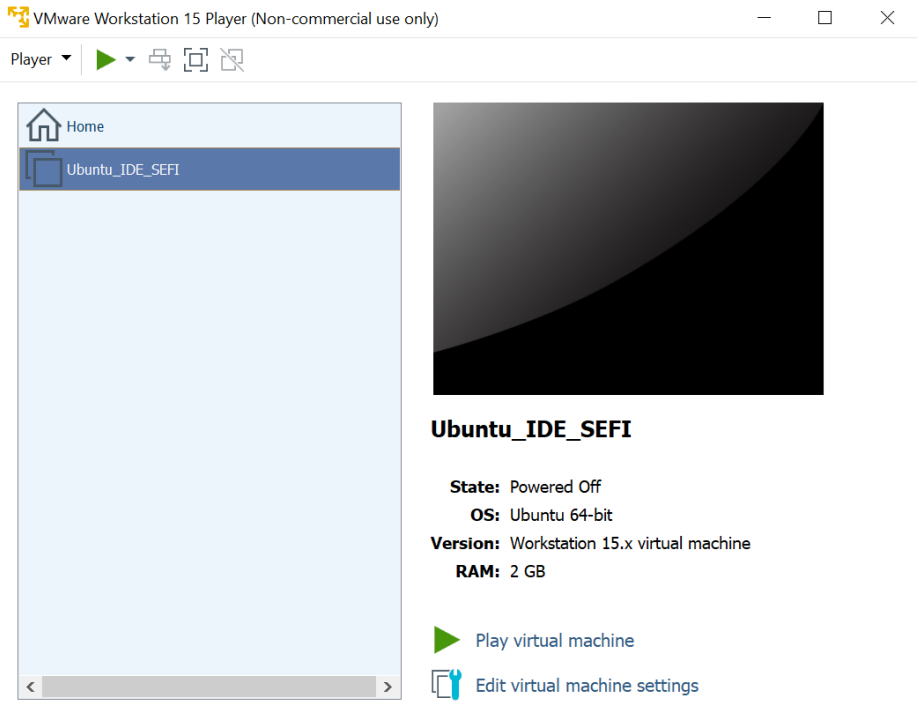


Abbildung 2 Aufrufen der Machine Settings auf der VMware Player Bedienoberfläche

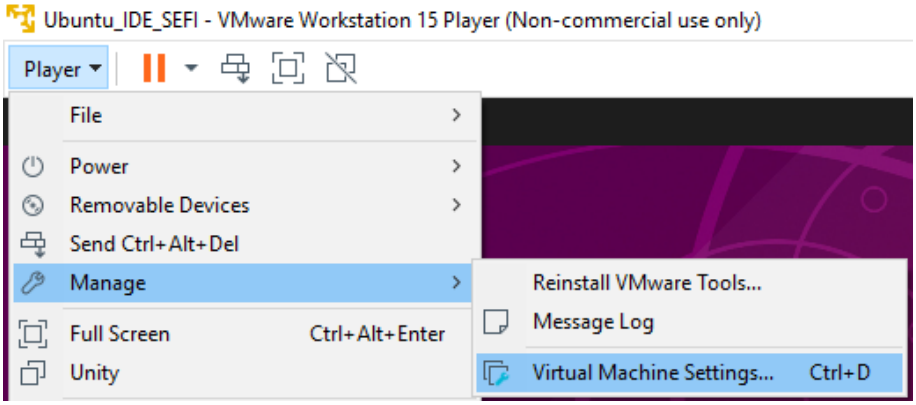


Abbildung 3: Aufrufen der Machine Settings über den Reiterpfad

Wählen Sie anschließend den Reiter *Options*. Unter *Shared Folders* müssen Sie *Always enabled* wählen. Mit *Add* können Sie den Ordnerpfad und –namen angeben, den Sie hinzufügen wollen. In diesem Praktium ist dies das Laufwerk H. Dies ist an Ihrer Sefi Nummer zu erkennen, die Sie am Anfang erhalten haben und mit der Sie sich an Windows angemeldet haben. Ab jetzt ist ihr persönliches Netzlaufwerk mit der virtuellen Maschine verbunden, sodass Sie lokal am Computer auf ihren Workspace zugreifen können. Als Workspace wird hierbei der Ordner bezeichnet, in welchem Sie später Ihre Projekte speichern werden. Achten Sie darauf, Ihre Projekte unbedingt auf

dem Netzlaufwerk zu speichern, da diese andernfalls im Verlauf des Praktikums durch andere Benutzer des PCs gelöscht werden könnten. Achten Sie darauf, dass Sie unter *Properties* den Haken bei *Read-only* **nicht** gesetzt haben.

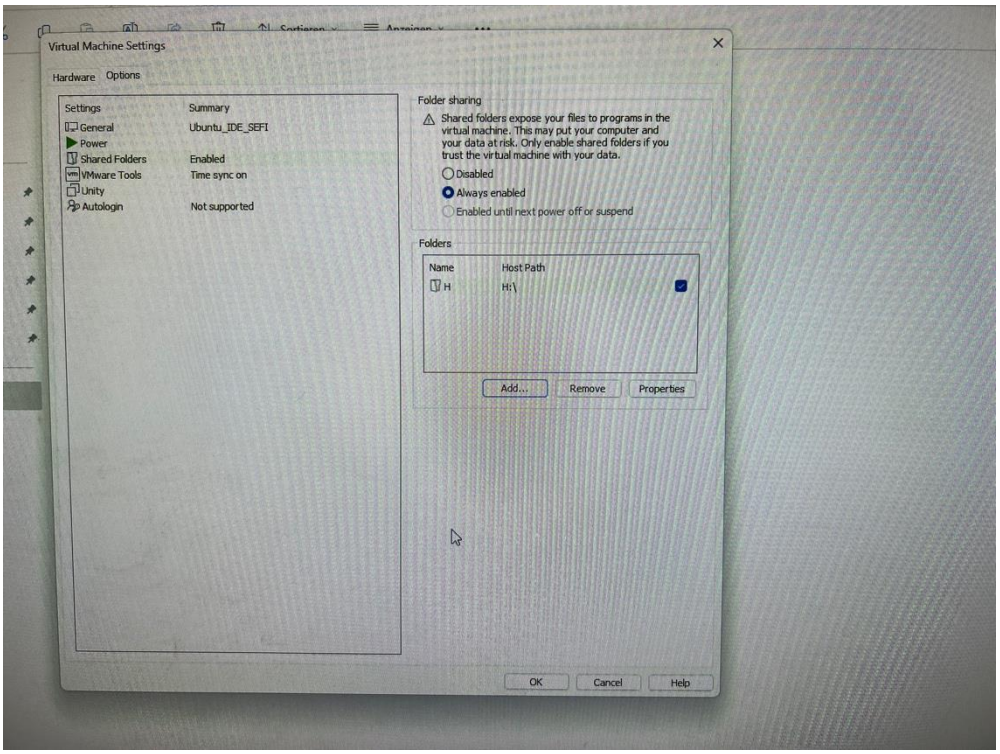


Abbildung 4: Netzlaufwerk einbinden

Durch Klicken auf *Play Virtual Machine* wird die Virtuelle Maschine gestartet. Beim ersten Start wird der VMware Player fragen, ob die Virtuelle Maschine kopiert oder verschoben worden ist. Hier *I copied it* auswählen.

Nautilus stellt in unserem Linuxsystem den „Dateiexplorer“ dar. Unter dem Pfad `\mnt\hgfs\...` finden Sie das eingebunden Netzlaufwerk. Diesen finden Sie, indem Sie unter *Andere Orte* → *Rechner* auswählen. In diesem Verzeichnis soll später der Workspace von Eclipse angelegt werden (Siehe Kapiel 1.7).

Falls die Ressourcen der VM bei Start noch von einem anderen Nutzer belegt sein sollten, erscheint eine Fehlermeldung, bei der Sie die Wahl haben, abzubrechen oder mittels *Take Ownership* die Ressourcen freizugeben. In der Regel sollten Sie die zweite Variante nutzen.

1.5 C/C++

C++, das Sie im Laufe der ersten drei Tage genauer kennenlernen werden, ist eine Weiterentwicklung der Programmiersprache C und aktuell eine der weltweit bedeutendsten Programmiersprachen, besonders im Ingenieursbereich. Einer der wichtigsten Unterschiede zu C ist der Wechsel vom imperativen zum objektorientierten Programmierparadigma. Was Objektorientierung bedeutet und wie sie umgesetzt wird, werden Sie in den weiteren Kapiteln kennenlernen.

1.6 Vom Code zum Programm

Um aus einem Quelltext ein lauffähiges Programm zu erzeugen, muss der Code jeder höheren Programmiersprache in Maschinencode übersetzt werden. Je höher die Programmiersprache, desto mehr Schritte erfordert diese Übersetzung. Im Folgenden betrachten wir die Schritte, die von C und C++ Code zum fertigen Maschinencode führen.

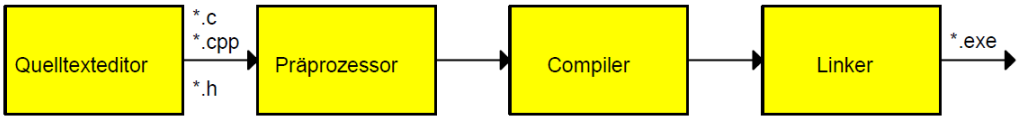


Abbildung 5: Schritte der Programmerstellung

1.6.1 Quelltexteditor

Im Quelltexteditor können Sie den Programmcode, wie bisher bekannt, schreiben. In C ist der Code aufgeteilt in `.c` und `.h` Dateien, in C++ in `.cpp` und `.h` Dateien.

Als Quelltexteditor kann man im Grunde jeden beliebigen Texteditor verwenden. Jedoch erleichtern Editoren mit Quelltextformatierungsfunktion erheblich die Arbeit. Die Quelltextformatierungsfunktion beinhaltet Folgendes:

- Syntaxhervorhebung: z.B. Datentypen in blau, etc.,
- Automatische Einrückungen,
- Automatisches Schließen von Klammern,
- Automatische Vervollständigung,
- Automatische Eingabekorrektur.

Der fertige Quelltext kann im nächsten Schritt an den Präprozessor weitergegeben werden.

Hinweis: Aufteilung des Codes in Implementierungs- und Headerdatei

Um die Übersichtlichkeit eines Quelltextes zu verbessern, teilt man diesen in Implementierungs- und Headerdateien auf.

Implementierungsdateien (Suffix: `.c` oder `.cpp`) enthalten:

- Funktionsdefinitionen,
- Einbindung der zugehörigen Headerdatei.

Headerdateien (Suffix: `.h`) enthalten:

- Funktionsdeklaration: Funktionskopf, der nur den Funktionsnamen, Rück- und Übergabeparameter beschreibt,
- Strukturdefinition,
- Einbindung anderer Header mit benötigten Funktionen und Bibliotheken.

1.6.2 Präprozessor

Der Präprozessor ist ein primitives Programm, welches den Quelltext verändert. Dabei führt der Präprozessor keine Analyse der Funktionalität des Codes durch, sondern geht nach dem Prinzip *Suchen und Ersetzen* vor.

Der Präprozessor hat unter anderen folgende Aufgaben:

- Ersetzen der Präprozessorkonstanten durch ihren Wert,
- Entfernen der Kommentare,
- Einbinden der Header- in die Implementierungsdateien.

Der Präprozessor wird durch Direktiven gesteuert, welche mit `#` beginnen. So kann man beispielsweise mit `#include` Headerdateien einbinden. Dabei muss allerdings darauf geachtet werden, dass es nicht zur Mehrfacheinbindung kommen kann. Dies kann durch den Einsatz von sogenannten Include-Guards mit Hilfe von `#define` sichergestellt werden.

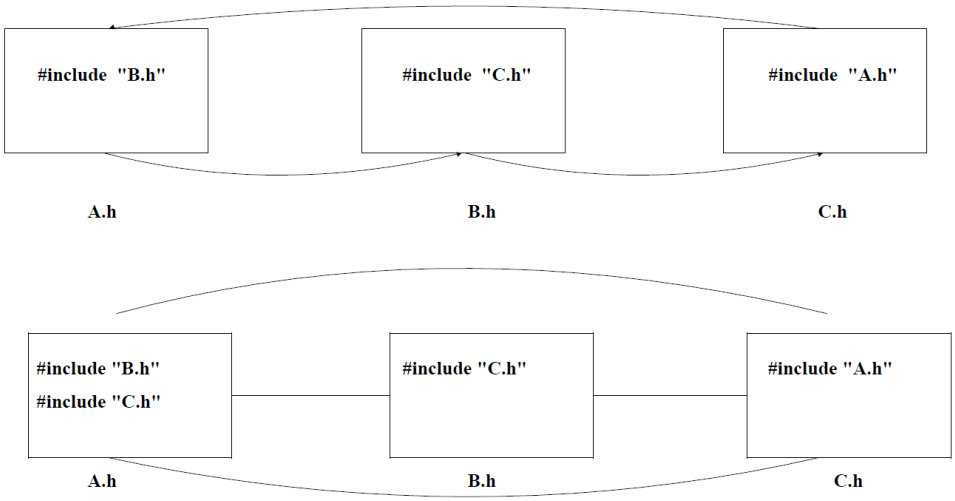


Abbildung 6: Endlosschleife durch Mehrfacheinbindung

Hinweis: Mehrfacheinbindung

Unter Mehrfacheinbindung versteht man die mehrfache Einbindung eines Headers durch den Präprozessor in eine Datei. Wie es dazu kommen kann, sehen Sie in Abbildung 6: Endlosschleife durch Mehrfacheinbindung.

Bei größeren Programmen ist es möglich, dass ein Header in mehrere Dateien eingebunden wird. So benötigt hier sowohl A.h als auch B.h den Header C.h. Da wiederum C.h den Header A.h einbindet, kommt es zu einer Endlosschleife. Der Präprozessor wird mit seiner Arbeit nie fertig. Mehrfacheinbindung verhindert man, indem man jedem Header Präprozessoranweisungen voranstellt (siehe Listing 1). Dadurch prüft der Präprozessor vor Einbindung eines Headers, ob dieser bereits definiert wurde, und übergeht gegebenenfalls die `include`-Anweisung.

```
1  // Ersetzen Sie FILENAME_H durch den Namen der Headerdatei
2  #ifndef FILENAME_H
3  #define FILENAME_H
4
5  // Hier steht Ihr Quelltext…
6
7  #endif
```

Listing 1: Direktive zum Verhindern der Mehrfacheinbindung

Nachdem der Präprozessor den Quelltext verändert hat, kann dieser an den Compiler weitergereicht werden.

1.6.3 Compiler

Der Compiler ist ein Computerprogramm, das die einzelnen zusammengebundenen Implementierungs- und Headerdateien in Maschinencode übersetzt. Dabei entstehen Objektdateien; `.o` ist die entsprechende Dateiendung.

Hinweis: Objektdateien

Objektdateien haben nichts mit Objektorientierung zu tun. Mit Objekten sind in diesem Fall Variablen und Funktionen gemeint.

Der Quelltext ist nun in Maschinencode übersetzt, jedoch sind die Objektdateien noch nicht ausführbar. Dafür müssen sie erst durch den Linker zu einem ausführbaren Code verarbeitet werden.

1.6.4 Linker

Der Linker ist ein Computerprogramm, welches die einzelnen Objektdateien zu einem ausführbaren Programm bindet. Zudem bindet er externe Bibliotheken in Form von Objektdateien ein, welche vorgefertigte Funktionen beinhalten. Das Programm liegt nach dem Linken (unter Windows) als `.exe` Datei vor und kann schließlich ausgeführt werden.

1.7 Eclipse

Im vorherigen Kapitel haben Sie die notwendigen Schritte (Editor, Präprozessor, Compiler, Linker) zur Erstellung eines lauffähigen Programms bereits kennengelernt. Diese Schritte können bei der Kommandozeilenarbeit (z.B. unter UNIX) nacheinander aufgerufen werden.

Im Gegensatz dazu handelt es sich bei Eclipse um eine sogenannte integrierte Entwicklungsumgebung. Sie erleichtert es Ihnen, aus dem Quelltext ein ausführbares Programm zu erstellen. Während des gesamten Praktikums werden Sie Eclipse in Ubuntu verwenden. Es ist sowohl für Ubuntu als auch für Windows kostenlos erhältlich.

Dieses Programm finden Sie auf dem Desktop der virtuellen Maschine.

1.7.1 Integrierte Entwicklungsumgebung

Eine integrierte Entwicklungsumgebung ist eine umfangreiche Software, die alle für die Softwareentwicklung nötigen Programme beinhaltet:

- Dateiverwaltung,
- Vorlagen für unterschiedliche Anwendungen,
- Quelltexteditor mit Quelltextauszeichnungsfunktion,
- Präprozessor,
- Compiler,
- Linker,
- Debugger.

Diese Zusammenstellung wird auch „Toolchain“ genannt und wird in den folgenden Kapiteln noch besprochen.

1.7.2 Dateiverwaltung

Wenn Eclipse das erste Mal startet, muss zunächst ein sogenannter Workspace-Ordner angegeben werden. Dort werden alle Projekte, die anschließend mit Eclipse

erstellt werden, gespeichert. Bitte verwenden Sie je nach Praktikumsart folgenden Speicherort für Ihren Workspace:

- Präsenzpraktikum: Ihr persönliches Netzlaufwerk, das sie bereits, wie im Kapitel 1.4 erläutert, eingebunden haben.
- Remotepraktikum: Wenn Sie das Praktikum über Remotezugriff bearbeiten, speichern Sie ihren Code lokal auf dem Desktop ihrer virtuellen Maschine. Die während des Praktikums geschriebenen Programmcodes sind nach erfolgreicher Abnahme aller Aufgaben im Moodlekurs zum C++ Praktikum hochzuladen. Der Upload muss für jedes einzelne Teammitglied erfolgen.

Kopieren Sie anschließend unbedingt den vorgefertigten Workspace Ordner, der sich bereits auf dem Desktop der VM befindet, an den oben beschriebenen Speicherort und wählen diesen als Eclipse-Workspace aus.

Jedes Programm erhält einen Projektordner. Dieser Projektordner umfasst sowohl Header- und Implementierungsdateien als auch weitere zur Ausführung des Programms notwendige Dateien, wie z.B. `.o`, `.d`, `.mk`, etc.

1.7.3 Vorlagen für die unterschiedlichen Anwendungen

Eclipse bietet Ihnen eine Auswahl an Projekttypen. Sie legen damit fest, für welches System Sie ein Programm erstellen und welche Toolchain verwendet wird.

Ihre erste Aufgabe besteht darin, ein neues Projekt anzulegen. Dieses Programm soll eine Konsolenanwendung sein, d.h. die Ein- und Ausgabe erfolgt über ein Konsolenfenster. Dafür verwenden wir ein leeres Linux GCC Projekt. Standardmäßig wird dafür ein neuer Projektordner angelegt. Bitte erstellen Sie für jede neue Aufgabe des Praktikums einen eigenen Projektordner sowie ein neues Projekt, außer es wird ausdrücklich nach einer Ergänzung eines bestehenden Programms gefragt.

1.7.4 Erstellung von ausführbarem Programmcode

Nachdem Sie die Dateien eingebunden haben, soll das Programm nun erstellt werden. Ausgehend vom Quelltexteditor durchläuft der Quelltext nacheinander

- Präprozessor,
- Compiler,
- Linker.

Wie bereits beschrieben, müssen Sie in Eclipse die einzelnen Programme nicht einzeln nacheinander aufrufen. Sie können das Programm mit einem Klick in der Menüleiste auf *Project* erstellen.

Hinweis: „Problems“-Fenster

Compilerfehler werden im Fenster „Problems“ angezeigt. Dieses Fenster enthält oft wertvolle Informationen über Fehler. Mit einem Doppelklick auf den Fehlereintrag können Sie an die Stelle im Code springen, wo der Compilerfehler aufgetreten ist.

In Tabelle 1 werden wichtige Menüpunkte im Menü „Project“ erklärt.

Menüpunkt	Erklärung
Build Project	Das Projekt wird in ein ausführbares Programm umgewandelt. Dabei werden alle geänderten `.cpp` Dateien neu erstellt.
Clean Project	Löscht die temporären Dateien der Projektmappe.
Properties	Öffnet ein Fenster, um Einstellungen beim Projekt vorzunehmen.

Tabelle 1: "Project" Menü

1.7.5 Debugging

Dieser Abschnitt dient dazu, Ihnen zu zeigen, was ein Debugger ist, wie er funktioniert und wie man ihn bedient.

Sobald ein Programm erfolgreich erstellt werden konnte, ist der Quelltext frei von Compilerfehlern. Dies muss jedoch nicht heißen, dass in dem Programm keine inhaltlichen Fehler vorhanden sind. Um diese zu finden, benutzen Sie den Debugger.

Der Debugger ist ein Werkzeug zur Fehlerdiagnose. Er wird benutzt, um den Programmablauf manuell zu steuern. Dadurch ist es zum Beispiel möglich, einen Quellcode Zeile für Zeile auszuführen, was die Fehlersuche erheblich vereinfacht.

Mit seiner Hilfe können Sie also manuell durch den Quelltext navigieren und dabei auch die Variablenwerte einsehen.

Die Arbeit mit dem Debugger läuft folgendermaßen ab:

1. Sie können Haltepunkte in jeder Zeile des Quelltextes definieren. Klicken Sie hierfür mit einem Rechtsklick auf den Zeilenanfang und wählen Sie im Kontextmenü „Toggle Breakpoint“ aus (siehe Abbildung 7). Es erscheint ein blauer Punkt.
2. Führen Sie jetzt das Programm im Debug-Modus aus. Klicken Sie dazu das Käfersymbol an (siehe Abbildung 8). Eclipse wechselt in die Debug Ansicht.
3. Der Debugger geht nun das Programm Zeile für Zeile durch.
4. Wenn der Programmzeiger den ersten Haltepunkt erreicht, hält die Programmausführung an. Der Programmzeiger steht nun vor der Zeile, in der Sie den Haltepunkt gesetzt haben.
5. Die Werte der einzelnen Variablen erfahren Sie im Fenster „Variables“.

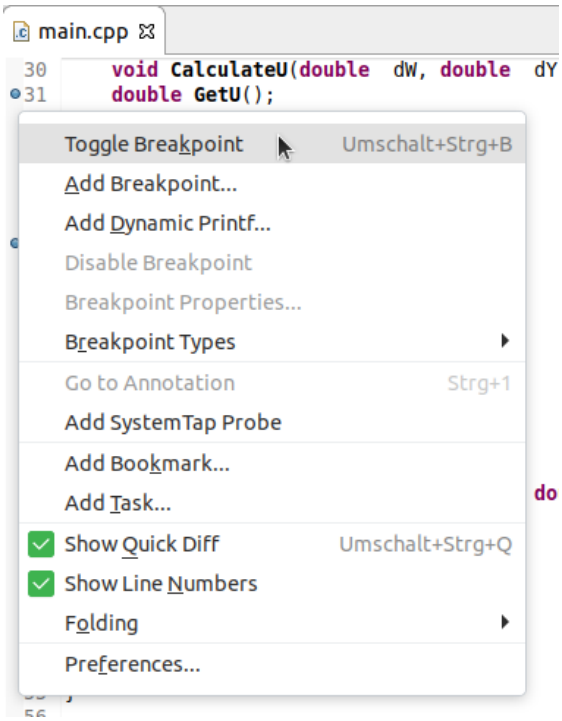


Abbildung 7: Breakpoint einfügen

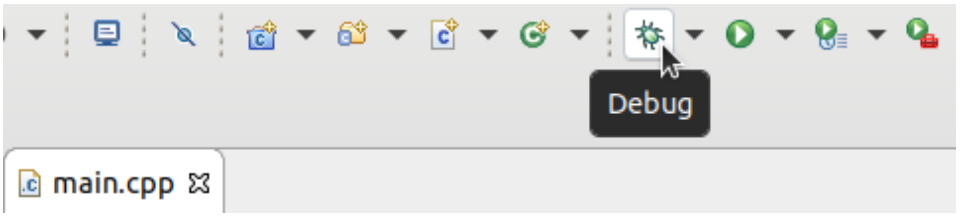


Abbildung 8: Debugging starten

Den Debugger steuern Sie über die Werkzeugleiste in der Debug-Ansicht (siehe Tabelle 2). Die wichtigsten dort aufgelisteten Funktionen werden im Folgenden näher beschrieben.

Der Umgang mit den Haltepunkten sollte Ihnen aus dem Grundstudium bekannt sein.

Um Variablen zu überwachen, gibt es das „Variables“-Fenster. Alle Variablen werden in dem Fenster mit ihren aktuellen Werten angezeigt.

Wenn Sie mit dem Debugging fertig sind, klicken Sie oben Rechts auf C/C++, um wieder in die Code-Ansicht zurück zu gelangen (siehe Abbildung 10).

Menüpunkt	Erklärung
Resume (F8)	Die Programmausführung wird aufgenommen und das Programm läuft bis zum nächsten Breakpoint oder bis zum Ende.
Terminate (Strg+F2)	Abbruch der Programmausführung.

Step Into (F5)	Die aktuelle Quelltextzeile wird ausgeführt und springt in die nächste Zeile.
Step Over (F6)	Es wird zur nächsten Quelltextzeile gesprungen.
Step Return (F7)	Beendet die Ausführung einer Methode.

Tabelle 2: Menüleiste im Debug-Modus

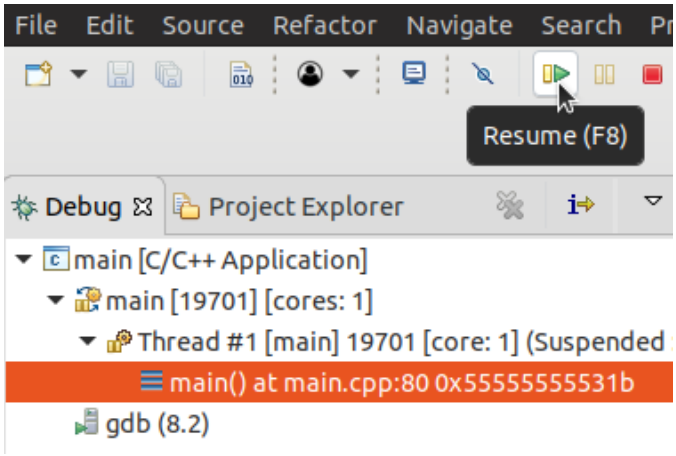


Abbildung 9: Werkzeugleiste im Debug-Modus

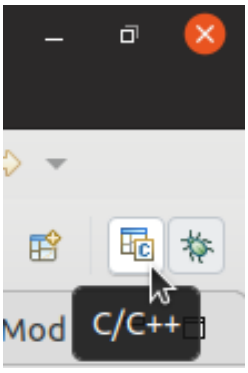


Abbildung 10: Wechsel zwischen Debug- und Programmieransicht

Hinweis: Debug- und Release-Modus

Damit der Debugger die zusätzlichen Funktionalitäten wie Quelltextinspektion, Variablenüberwachung und Haltepunkte umsetzen kann, muss der Compiler zusätzliche Informationen generieren und ablegen. Außerdem führt der Compiler keine Optimierungen durch, damit auch jede Programmzeile im ausführbaren Programm vorhanden ist. Dies erhöht den Speicherbedarf und verringert die Effizienz. Aus diesem Grund sollten fertige Programme immer im Release-Modus erstellt werden. Zwischen Debug- und Release-Modus können Sie in der Werkzeugleiste umschalten.

Hinweis: Verwendung des lokalen Compilers

Bei der Verwendung von Eclipse mit der in diesem Praktikum verwendeten Virtuellen Maschine kann es bei der Ausführung des in C++ programmierten Programms (Mit Klick auf *Run*) dazu kommen, dass Sie wählen müssen, was für eine Applikation Sie erstellt haben. Sie haben hierbei die Wahl zwischen *C/C++ Container Application* und *Local C/C++ Application*. Wählen Sie hierbei die ***lokale Application***.

1.7.6 Erstellen eines C++ Projekts

Um ein neues C++ Projekt zu erstellen, klicken Sie auf *File* → *New* und wählen New *C++Project*. Falls Sie aufgefordert werden, auszuwählen, welche Art von Projekt (z.B. C Project, Android Build, …) Sie erstellen möchten, wählen sie *C++ Managed Build*. Nach der Eingabe eines Projektnamens wählt man *Hello World C++ Project.* Anders als bei *Empty Project* wird hier gleich eine Ordnerstruktur aus *src* und *includes* miterzeugt. Als Toolchain wird *Linux GCC* ausgewählt. Nachdem man drei Mal auf *Next* und abschließend auf *Finish* geklickt hat, wird das Projekt erzeugt. Durch einen Rechtsklick auf das Projekt im Project Explorer kann unter *New* ein neues *Source File* bzw. *Header File* erzeugt werden. Dabei ist es wichtig, die entsprechende Dateiendung **.cpp* bzw. **.h* zum selbst gewählten Dateinamen hinzuzufügen. Um den Code kompilieren zu können ist es notwendig, exakt eine *main()* Funktion zu definieren. Diese markiert den Einstiegspunkt des Programms und wird nach dessen Start automatisch aufgerufen.

1.7.7 Tipps und Tricks

Für einen effizienteren Umgang mit Eclipse geben wir Ihnen an dieser Stelle noch einige kurze Hinweise.

Wenn Sie mit der rechten Maustaste auf eine Funktion oder Variable klicken, öffnet sich ein Kontextmenü, welches viele nützliche Funktionen anbietet. Am häufigsten wird dabei „Open Declaration (F3)“ benötigt. Durch diese Funktion springt man direkt zur Deklaration der ausgewählten Methode oder Variable. Außerdem kann unter „References“ angezeigt werden, wo die auswählte Methode/Variable eingesetzt wird.

Der Funktions- und Methodenbrowser befindet sich am rechten Rand des Codeeditors. Dieser listet alle in der aktuellen Implementierungsdatei vorhanden Funktionen auf und ermöglicht es Ihnen durch Anklicken sehr schnell zur Funktionsdefinition zu springen.

Sollte beim Kompilieren oder Ausführen eines Programms die Meldung „**Permission denied**“ in der Konsole von Eclipse ausgegeben werden, so liegt dies meist daran, dass das eigene Programm noch immer im Hintergrund unsichtbar geöffnet ist. Um

das Programm zu beenden, drücken Sie Strg+Esc, suchen im anschließenden Fenster nach dem betreffenden Projektnamen und beenden dieses.

Um ganze Code-Blöcke auszukommentieren, kann die Tastenkombination Strg+Shift+7 verwendet werden.

Zum automatischen Formatieren eines Code-Blocks kann die Tastenkombination Strg+Shift+F verwendet werden.

Die Tabelle 3 zeigt Tastaturkürzel, die bei Nutzung den Umgang mit Eclipse vereinfachen können:

Automatisch Aus-/Entkommentieren eines Codeblocks:	*Strg + Shift + 7*
Autoformat des Codes:	***Strg + Shift + F***
Speichern der aktiven Datei:	*Strg + S*
Speichern aller geöffneten Dateien:	*Strg + Shift + S*
Build **aller** Projekte:	*Strg + B*
Build and Run:	*Strg + F11*
Aktive Zeile entfernen:	*Strg + D*
Zeile verschieben:	*Alt + Pfeiltaste*
Quick-Menu für die Dateien eines Projekts:	*Strg + E*
Ansicht Maximieren/Demaximieren:	*Strg + M*

Tabelle 3 Eclipse Tastenkombinationen

Der Profi-Tipp für den fortgeschrittenen Programmierer

Speichern und Builden/Kompilieren nicht vergessen! Einige Fehler lassen sich hiermit beseitigen, da die IDE die letzten Änderungen noch nicht übernommen hat.

1.8 Grundlagen

In diesem Kapitel werden zwei grundlegende Elemente von C++ eingeführt. Zum einen Variablen und Datentypen, die sehr der Umsetzung in C ähneln. Zum anderen die Ein- und Ausgabe auf der Konsole, die in C++ verändert wurde und sich grundlegend von der Ein- und Ausgabe in der Programmiersprache C unterscheidet.

1.8.1 Variablen und Datentypen

1.8.1.1 Datentypen

Variablen in C++ speichern einen Wert eines bestimmten Datentyps. Einige Grunddatentypen in C++ (auf einem gängigen 32-Bit System sowie den meisten ARM-Prozessoren wie z.B. im Raspberry Pi) sind in Tabelle 4 aufgelistet.

Typ	Speicherplatz	Wertebereich (decimal)	
`Bool`	1 Byte	True (1), false (0)	
`Char`	1 Byte	-128 bis +127	
`unsigned char`	1 Byte	0 bis 255	
`Short`	2 Byte	-32.768 bis +32.767	
`unsigned short`	2 Byte	0 bis 65.535	
`int`	4 Byte	-2.147.483.648 bis +2.147.483.647	
`unsigned int`	4 Byte	0 bis 4.294.967.295	
`long`	4 Byte	-2.147.483.648 bis +2.147.483.647	
`unsigned long`	4 Byte	0 bis 4.294.967.295	
`float`	4 Byte	±3,4* 10^38	Genauigkeit: 6 Stellen
`double`	8 Byte	±1,7 *10^308	Genauigkeit: 15 Stellen
`long double`	10 Byte	±1,1* 10^4932	Genauigkeit: 19 Stellen

Tabelle 4: Datentypen in C++

1.8.1.2 Deklaration und Initialisierung von Variablen

In C++ müssen Variablen deklariert werden, bevor sie verwendet werden können. Damit wird der Compiler angewiesen, entsprechend Speicherplatz für die Variable zu reservieren. Erst nach der Initialisierung, d.h., wenn der Variable das erste Mal ein Wert zugewiesen wurde, hat sie sicher einen definierten Wert. Eine Variable wird wie folgt deklariert und initialisiert:

```
1  int iNumber;                    // Deklaration einer Variable vom Typ int
2  iNumber = 10;                   // Initialisierung der Variable
3  float fNumber = 2.8;            // Deklaration und Initialisierung einer Variable vom Typ float
4  char chValue = 'A';             // Deklaration und Initialisierung einer Variable vom Typ char
```

Listing 2: Variablendeklaration und Initialisierung

Um eine hohe Qualität, leichtere Lesbarkeit und allgemeine Einheitlichkeit des geschriebenen Codes zu ermöglichen, werden für die Bezeichnung der Variablen sogenannte naming conventions verwendet. Eine kurze Zusammenfassung ist im Anhang 5.2 zu finden. Nutzen Sie diese für das vor Ihnen liegende Praktikum.

1.8.1.3 Gültigkeitsbereich von Variablen

Variablen sind im Allgemeinen nur in dem Block gültig, in dem sie deklariert wurden. D.h. um eine Variable in einem bestimmten Block nutzen zu können, muss sie in diesem oder in einem umschließenden Block zuvor definiert werden. In C++ wird ein Block durch geschweifte Klammern – `{` und `}` – gekennzeichnet. Variablen, die außerhalb einer Funktion (oder Klasse) definiert werden, sind globale Variablen, die allen Funktionen in demselben Namespace zugänglich sind.

```
1   int iNumber = -4;            // iNumber existiert im if-und else-Block
2   if(iNumber >= 0)
3   {
4      int iAbs;                 // iAbs existiert nur innerhalb des if-Blocks
5      iAbs= iNumber;
6   }
7   else
8   {
9      iAbs= -iNumber;           // FEHLER: iAbs in diesem Block nicht definiert!
10  }
```

Listing 3: Gültigkeitsbereich von Variablen

1.8.2 Ein- und Ausgabe auf der Konsole

In C++ werden die aus C bekannten `printf` und `scanf` Methoden der Bibliothek `stdio.h` nicht mehr verwendet. Stattdessen benutzt man nun die Bibliothek `iostream`. Diese muss durch die Präprozessoranweisung `#include <iostream>` eingebunden werden. Die wichtigsten Funktionen von `iostream` sind:

`std::cout` — Ermöglicht die Ausgabe auf der Konsole. Die auszugebenden Inhalte (Text, Variablen, etc.) übergibt man der Konsole mithilfe des Schiebeoperators `<<` wie folgt:

`std::cout << <Ausgabe> << std::endl;`

Durch `std::endl` beginnt die nächste Ausgabe in einer neuen Zeile.

`std::cin` — Ermöglicht das Einlesen von Benutzereingaben aus der Konsole mit Hilfe des Schiebeoperators `>>`:

`std::cin >> <Variablenname>;`

`std::cin` besitzt noch einige nützliche Eigenschaften, die es ermöglichen, Fehleingaben des Benutzers abzufangen. Wenn die letzte Eingabeoperation von `cin` aufgrund eines Problems (in Bezug auf die interne Logik der Operation selbst) fehlschlägt, so wird das Failbit gesetzt.

`std::cin.fail()` — gibt an, ob das Failbit gesetzt ist (`true`, wenn Failbit tatsächlich gesetzt ist).

`std::cin.clear()` — löscht ein zuvor gesetztes Failbit von `cin`.

`std::cin.ignore(m,'\n')` ignoriert maximal m Zeichen im Stream von `cin` bis zum ersten Auftreten des Zeichens `\n` (Zeilenumbruch).

1.8.2.1 Abfangen von Fehleingaben mit std::cin

Gemeinsam können die drei zuvor genannten Attribute von `cin` verwendet werden, um Falscheingaben eines Benutzers abzufangen. Folgender Code stellt beispielhaft sicher, dass eine korrekte Zahl eingegeben wurde, wenn eine Ganzzahl eingelesen werden soll:

```
1  int iNumber = 0;
2  std::cin >> iNumber;
3  while(std::cin.fail())                    // Gehe in Schleife solange Failbit gesetzt
4  {
5     std::cout << "Falscheingabe: Bitte Ganzzahl eingeben" << std::endl;
6     std::cin.clear();                      // Setze Failbit zurück
7     std::cin.ignore(1000, '\n');           // Verwerfe bisherige Eingabe bis zu einem Zeilenumbruch
8     std::cin >> iNumber;                   // Versuche Zuweisung erneut
9  }
```

Listing 4: Abfangen von Fehleingaben

1.9 Kontrollstrukturen

In diesem Kapitel wiederholen wir noch einmal die Kontrollstrukturen. Diese sollten Ihnen schon alle aus dem Grundstudium bekannt sein, da es diesbezüglich kaum Unterschiede zwischen C und C++ gibt. Kontrollstrukturen steuern den Ablauf eines Computerprogramms, indem sie die streng lineare Abarbeitung des Codes aufbrechen. Man unterscheidet zwischen Verzweigungen und Schleifen.

1.9.1 Verzweigungen

Verzweigungen bestehen aus einer oder mehreren Bedingungen und unterschiedlichen Codeblöcken. Erreicht die Programmausführung eine Verzweigung, wird zuerst die vorgegebene Bedingung überprüft. Danach wird, je nach Ergebnis, der entsprechende Codeblock ausgeführt.

1.9.1.1 if-Verzweigung

Die `if`-Verzweigung hat die in Listing 5 aufgeführte Syntax. Der `else`-Zweig kann ggf. weggelassen werden.

```
1  if(<Bedingung>)
2  {
3     // Anweisungen für wahre Bedingung
4  }
5  else      // Optionaler else-Zweig
6  {
7     // Anweisungen für falsche Bedingung
8  }
```

Listing 5: `if`-Verzweigung Syntax

Listing 6 zeigt ein Beispiel für eine `if`-Verzweigung.

```
1  if(iPieceNumber == 0)
2  {
3     printError("No piece in stock.");
4  }
5  else
6  {
7     sendPiece();
8  }
```

Listing 6: `if`-Verzweigung Beispiel

Hinweis: Vergleichs- und Zuweisungsoperator

Achten Sie darauf, Vergleichsoperator (==) und Zuweisungsoperator (=) nicht zu verwechseln.

1.9.1.2 Bedingter Ausdruck

In C++ ist es ebenfalls möglich, `if`-Verzweigungen verkürzt als einen bedingten Ausdruck umzusetzen. Diese Art der Programmierung sollten Sie jedoch vermeiden, da dadurch die Quelltextstruktur sehr unübersichtlich werden kann.

Die Syntax des bedingten Ausdrucks sehen Sie in Listing 7.

```
1  <Bedingung> ? <Anweisung für wahre Bedingung> : <Anweisung für falsche Bedingung>
```

Listing 7: Syntax Bedingte Anweisung

Ein Beispiel der Verwendung gibt Listing 8:

```
1  int iNumber = 5;
2  iNumber = (iNumber > 0) ? iNumber : (-1) * iNumber;    // gibt absoluten Wert von iNumber zurück
```

Listing 8: Verwendung Bedingte Anweisung

1.9.1.3 switch-Verzweigung

Die `switch`-Verzweigung kommt zum Einsatz, wenn, je nach Wert einer Variablen, eine andere Aktion ausgeführt werden soll. Sie ermöglicht die Auswahl zwischen beliebig vielen Alternativen. Damit kann sie als eine geschachtelte `if-else` Anweisung angesehen werden. In Listing 9 ist die Syntax der `switch`-Verzweigung aufgeführt.

```
1  switch(<Variable>)
2  {
3     case <Wert>:
4           // Anweisung
5           break;
6     case <Wert>:
7           // Anweisung
8           break;
9     case <Wert>:
10          // Anweisung
11          break;
12    // weitere Case-Abfragen
13    default:
14          // Anweisung
15 }
```

Listing 9: `switch`-Verzweigung Syntax

Hinweis: `break`

Die Anweisung `break` bedeutet das Verlassen der Verzweigung und muss nach jeder Anweisung aufgerufen werden, damit nicht zusätzlich die darauffolgenden Cases ausgeführt werden.

Sie können ein Beispiel für die `switch`-Verzweigung dem Listing 10 entnehmen.

```
1   switch(iPieceNumber)
2   {
3       case 1:
4               printPieceNumber();
5               break;
6       case 2:
7               selectFirstPiece();
8               printPieceNumber();
9               break;
10      default:
11              sendErrorMessage();
12  }
```

Listing 10: `switch`-Verzweigung Beispiel

1.9.2 Schleifen

Schleifen bestehen aus einem Codeblock und einer Wiederholungsbedingung. Der Codeblock wird so lange ausgeführt, wie die Wiederholungsbedingung erfüllt ist.

1.9.2.1 for-Schleife

Im Unterschied zu früheren Versionen von C kann bei C++ auch erst im Schleifenkopf der Zähler initialisiert werden.

Ein Beispiel für die `for`-Schleife sehen Sie in Listing 11.

```
1   for(int i = 0; i < 5; i++)
2   {
3       std::cout << i << std::endl;
4   }
```

Listing 11: Beispiel einer `for`-Schleife

1.9.2.2 while-Schleife

Bei der `while`-Schleife wird die Wiederholungsbedingung vor jedem Durchlauf geprüft, siehe Listing 12.

```
1   while(<Wiederholungsbedingung>)
2   {
3       // Anweisungen
4   }
```

Listing 12: Syntax der `while`-Schleife

1.9.2.3 do-while-Schleife

Bei der `do-while`-Schleife wird die Wiederholungsbedingung, anders als bei `while`-Schleife, erst nach jedem Durchlauf geprüft und die Schleifenausführung

gegebenenfalls abgebrochen, wie in Listing 13 gezeigt. Deshalb eignet sie sich besonders gut für Schleifen, die mindestens einmal durchlaufen werden müssen.

```
1  do
2  {
3     // Anweisungen
4  }
5  while(<Wiederholungsbedingung>);
```

Listing 13: Syntax der do-while-Schleife

Hinweis: do-while-Schleife

Vergessen Sie bei `do-while`-Schleifen nicht den Strichpunkt nach der Schleifenbedingung, da dies andernfalls zu Fehlern führt.

1.9.3 Funktionen

```
1  double  dSumme( int y, float z )
2  {
3     // Funktion mit Übergabe - und Rückgabewerten
4     return y + z + 3;
5  }
6
7  int main()
8  {
9     // main function
10    double dNumber = dSumme( 2 , 2.5 );
11    return 0;
12 }
```

Listing 14: Syntax einer Funktion mit Rückgabewert und ein beispielhafter Aufruf

Aufgabe: Taschenrechner

Im Folgenden soll ein Taschenrechner erstellt werden, der die vier Grundrechenarten ausführen kann. Der Taschenrechner arbeitet nach dem folgenden Prinzip: Einlesen der ersten Zahl, dann des Operators für die Rechenart und dann der zweiten Zahl. Achten Sie auf Kommazahlen und auf unerlaubte Rechenoperationen.

a) Schreiben Sie ein Programm, das eine vom Benutzer eingegebene Zahl einliest und auf Gültigkeit überprüft. Bei ungültiger Eingabe soll der Benutzer die Möglichkeit erhalten, erneut einen Wert einzugeben.

b) Als Nächstes wird der Operator eingelesen. Benutzen Sie den Datentyp *char*, um einen Operator einzulesen (+, –, *, /). Berücksichtigen Sie Falscheingaben.

Lesen Sie anschließend die zweite Zahl ein (achten Sie auch weiterhin auf Fehleingaben) und führen Sie anschließend die gewünschte Berechnung durch.

Achten Sie auf sinnvolle mathematische Berechnungen.

Aufgabe: Maya Zahlen

Das Zahlensystem der Maya wurde zur Angabe von teilweise sehr großen Zahlen für kalendarische Angaben und Berechnungen verwendet. Die Zählweise basierte dabei nicht auf dem uns geläufigen Dezimalsystem (Zehnersystem), sondern auf dem Vigesimalsystem (Zwanzigersystem).

Schreiben Sie ein Programm, das eine beliebige Zahl aus dem Zehnersystem einliest, sie anschließend in das Zahlensystem der Maya umrechnet und ausgibt. (Hinweis: Sie können den Modulo-Operator verwenden: `%`) Beispiel:

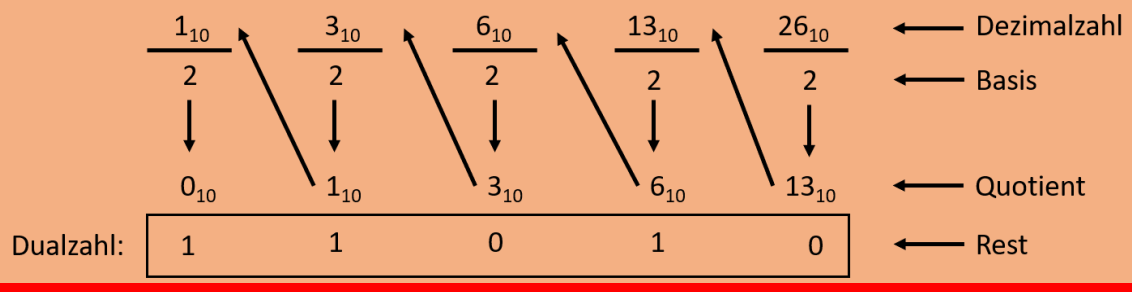
Zahl in Zehnersystem: 440 $(4*10^2 + 4*10^1 + 0*10^0)$

Zahl in Zwanzigersystem: 120 $(1*20^2 + 2*20^1 + 0*20^0)$

Mögliche Ausgabe: 0
2
1 von unten nach oben gelesen 120.

Hinweis: Horner-Schema

Verwenden Sie das Horner-Schema:



Aufgabe: Verfahren von Heron

Heron von Alexandria hat ein Verfahren entwickelt, um die Quadratwurzel einer Zahl zu berechnen. Die Iterationsvorschrift zur Ermittlung der Quadratwurzel lautet

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2},$$

wobei a die Zahl ist, deren Quadratwurzel gesucht wird und x_0 ein beliebiger Startwert ungleich 0.

a) Schreiben Sie eine Funktion `heron_ite(…)`, welche Das Heron-Verfahren iterativ ausführt. Die Anzahl an Iterationen beträgt 10.

b) Schreiben Sie eine `main`-Funktion, die eine Zahl vom Benutzer einliest und deren Quadratwurzel ausgibt (d.h. die Funktion `heron_ite(…)` aufruft und deren Ergebnis in die Konsole schreibt).

1.10 String

Strings können in C++ wie gewöhnliche Datentypen verwendet werden. Dafür muss die externe Bibliothek `string` mit folgendem Befehl eingebunden werden: `#include <string>`.

1.10.1 Methoden und Operatoren in `<string>`

Die Bibliothek `<string>` bietet umfangreiche Möglichkeiten zur Manipulation von Zeichenketten. Tabelle 5 listet einige auf. Hierbei sind `s1` und `s2` vom Typ `string` sowie `n`, `pos` und `x` vom Typ `int`. `x` kann ebenfalls vom Typ `char` sein.

Methode / Operator	Funktion
`char ch = s1[i];`	Ist eine Referenz auf das `i`-te Zeichen von `s1`. Achtung: Die Referenz ist vom Typ `char`, da sie ein einzelnes Zeichen darstellt.
`int n = s1.size();`	`n` ist die Anzahl der Zeichen in `s` als `int`.
`s1.append(n, x);` `s1.append(s2);`	Fügt `x` `n`-Mal am Ende von `s1` an. Fügt `s2` am Ende von `s1` an.
`s1.replace(pos, n, s2)`	Ersetzt die Zeichen im Intervall [`pos`, `pos+n`] in `s1` durch `s2`. `s1` und `s2` sind dabei vom Typ `string`.
`int n = s1.compare(s2);`	Vergleicht `s1` und `s2`: `n` ist `0`, wenn `s1` und `s2` identisch; `n` kleiner `0`, wenn `s1<s2`
`int pos =s.find(x);`	Findet die erste Position von `x` in `s`.
`std::string s2 =` `  s1.substr(pos,n);`	Kopiert die Zeichen im Intervall [`pos`, `pos+n`] von `s1` nach `s2`.
`int x = stoi(s);`	Konvertiert den `string s` zu einem `int`.

Tabelle 5: Einige Member-Methoden der `string`-Klasse

Listing 15 zeigt die Verwendung einiger Methoden aus der Bibliothek `<string>`.

```cpp
#include <string>
#include <iostream>

int main()
{
    int n = 0, pos = 0;
    std::string str = "Ein Beispielstring";          // Deklaration und Initialisierung eines strings

    for(int i = str.size(); i >= 0; i--)
    {
        std::cout << str[i] << std::endl;            // Zugriff auf einzelne Zeichen via []-Operator
    }

    int m = str.find('B');                           // Funktion find(x) im Einsatz
    str.replace(m, 8, "");                           // Funktion replace(pos,n,s2) im Einsatz
    std::cout << str << std::endl;
    return 0;
}
```

Listing 15: Einsatz der Bibliothek `<string>`

Aufgabe: Palindrome

Ein Palindrom ist ein Wort, das von rechts wie von links gelesen gleich lautet, wie z.B. das Word „rotor“.

a) Erstellen Sie eine Funktion, die testet, ob ein übergebenes Wort ein Palindrom ist. Es müssen nur Wörter in Kleinbuchstaben berücksichtigt werden. Der Operator `[]` (siehe Tabelle 5) kann für den Vergleich einzelner Zeichen verwendet werden. (Beachten Sie, dass der Rückgabewert von `[]` vom Typ `char` ist.)

b) Schreiben Sie ein Programm, das ein Wort einliest und zurückgibt, ob es ein Palindrom ist.

Aufgabe: Buchstaben und Wörter

Nach eienr Sutide der Cambrigde Uinverstiaet, ist es eagl in wlehcer Reiehnfogle die Bchustbaen in Woeretrn vokrmomen. Es ist nur withcig, dsas der ertse und lettze Bchusatbe an der ricthgien Stlele ist. Der Rset knan taotl falcsh sien und man knan es onhe Porbelme leesn. Das ist so, wiel das mneshcilche Geihrn nciht jdeen Bchustbaen liset, sodnern das Wrot als gaznes.

a) Schreiben Sie eine Funktion, die nach dem Schema des obigen Textes in einem übergebenen Wort die Reihenfolge der Buchstaben verändert. Dabei sollen immer zwei benachbarte Buchtstaben getauscht werden (Beispiel -> Biepseil).

b) Ein weiteres Phänomen ist, dass viele Wörter im Deutschen auch ohne Vokale gelesen werden können. Schreiben Sie eine weitere Funktion, die alle Vokale aus einem Wort entfernt.

c) Schreiben Sie eine `main`-Funktion, die Wörter vom Benutzer einliest und sie einmal mit verkehrter Buchstabenreihenfolge und einmal ohne Vokale ausgibt.

Tipp: Methoden für Strings, siehe Tabelle 5

1.11 Pointer

In diesem Kapitel wiederholen Sie, was man unter Pointern versteht und warum und wie man sie verwendet.

1.11.1 Definition

Ein Pointer ist eine Variable, die die Speicheradresse einer anderen Variablen enthält. Ein Pointer gibt somit an, an welcher Stelle im Hauptspeicher die Variable liegt. Man sagt, der Pointer referenziert diese Variable.

Pointer können aber prinzipiell an jede beliebige Stelle des Hauptspeichers zeigen und auf diese Weise sowohl ungewollte als auch gewollte Manipulationen an den aktiven Programmen im Hauptspeicher ermöglichen.

Kennt man die Adresse einer Variablen im Hauptspeicher, kann man auch auf ihren Wert zugreifen. Diesen Vorgang nennt man dereferenzieren.

1.11.2 Gründe für den Einsatz von Pointern

Drei wichtige Gründe für den Einsatz von Pointern sind:

- In C und C++ werden Parameter an Funktionen als Kopien übergeben. Die Kopien werden nach abgeschlossener Funktionsausführung gelöscht. Ein Verändern des Übergabeparameters im Gültigkeitsbereich der aufrufenden Funktion durch Veränderung der Kopie des Parameters in der aufgerufenen Funktion ist somit nicht möglich.

  Möchte man nun die Übergabeparameter in der Funktion verändern, so kann man der Funktion einen Pointer auf die Variable übergeben. Dies ermöglicht den Zugriff auf den Variablenwert über die Adresse, die der Pointer enthält. So kann man über die Adresse der Variablen im Hauptspeicher sowohl auf deren Inhalt zugreifen, als auch diesen ändern.
- Arrays kann man in C und C++ generell nur mit Hilfe von Pointern ansprechen.
- Pointer erlauben den Zugriff auf dynamisch reservierte Speicherbereiche (siehe Kapitel 1.13).

1.11.3 Syntax

In diesem Abschnitt beschäftigen wir uns kurz mit der Syntax zur Arbeit mit Pointern.

1.11.3.1 Deklaration und Initialisierung

Einen Pointer deklarieren Sie durch Angabe des Datentypens der Variablen, auf die er zeigt, gefolgt von dem Stern-Operator und den Namen des Pointers. Beispiele hierfür sehen Sie in Listing 16, Zeile 2 und 5.

Die Adresse einer bestehenden Variable erhalten Sie durch voranstellen des &-Operators vor den Variablennamen (vgl. Zeile 8).

```
1   // Deklaration eines Pointers auf einen char
2   char* pchLabelList;
3
4   // Deklaration und Initialisierung eines Pointers auf einen float mit der Adresse im Hauptspeicher
5   float* pfWidth = 0x0013FE94;
6
7   // Initialisierung eines Pointers auf die Adresse einer int Variable
8   int* piPieceNumber = &iPieceNumber;
```

Listing 16: Pointer-Deklaration

1.11.3.2 Dereferenzierung

Um mit Hilfe eines Pointers auf die Werte im Hauptspeicher zuzugreifen, die der Pointer referenziert, kann man zwei Operatoren benutzen, dargestellt in Listing 17.

- Erstens den Stern-Operator `*`. Er steht vor dem zu dereferenzierenden Pointer, vgl. Zeile 2.
- Zweitens den Klammern-Operator `[]`, der dahinter steht, vgl. Zeile 3.

```
1   // Zugreifen auf den Wert  den piPieceNumber referenziert
2   int iNumber = *piPieceNumber;
3   iNumber = piPieceNumber[];
```

Listing 17: Dereferenzierung

Wenn Sie auf den Wert der Hauptspeicheradresse vor oder nach der Adresse zugreifen möchten, die der Pointer referenziert, benutzen Sie die in Listing 18 aufgeführte Syntax.

```
1   // Zugreifen auf den ersten Buchstaben des Labels
2   char chLetter = *pchLabel;
3   chLetter = pchLabel[0];
4   chLetter = pchLabel[];
5
6   // Zugreifen auf den fünften Buchstaben
7   chLetter = *(pchLabel+4);
8   chLetter = pchLabel[4];
```

Listing 18: Pointerarithmetik

1.11.3.3 Parameterübergabe per Pointer

Wie bereits erwähnt, werden Pointer sehr häufig für die Übergabe von Parametern in Funktionen verwendet. In Listing 19 finden Sie ein Anwendungsbeispiel.

```
1   // Signatur der Funktion, die den Pointer auf ein Array mit Nutzdaten
2   // erhält, die den Mittelwert daraus berechnet
3   void calculateMean(float* pfMean, int* piData, int iSize);
4   // …
5
6   int main()
7   {
8       float fMean = 0.0;
9       int piData[] = {1, 2, 3, 4, 58, 5, 6, 1, 98, 3};
10      // Aufruf der Funktion
11      calculateMean(&fMean, piData, 10);
12      return 0;
13  }
```

Listing 19: Funktionsdeklaration mit Pointer

Aufgabe: Quadratische Funktionen

a) Schreiben Sie eine Funktion, welche die (reellen) Lösungen einer quadratischen Gleichung berechnet (Mitternachtsformel). Der Rückgabewert der Funktion soll angeben, ob die Gleichung eine Lösung hat. Als Argumente bekommt die Funktion die Koeffizienten a, b, c und zwei Pointer auf die Lösungen (warum?).

b) Schreiben Sie eine main-Funktion, die die Koeffizienten a, b, c einliest und die reellen Lösungen der zugehörigen quadratischen Gleichung ausgibt. Testen Sie die Funktion mit

$$2x^2 + 5x + 2 = 0 \quad x_1 = -0.5\,,\ x_2 = -2.0$$
$$x^2 - 6x + 9 = 0 \quad x_1 = 3.0\,,\ x_2 = 3.0$$
$$x^2 + 10 = 0 \quad keine\ Lösung$$

Beachten Sie die Hinweise.

Hinweis: Deklaration von Pointern

Die folgenden Schreibweisen für die Deklaration von Pointern sind identisch:

```
int* piNumber;
int *piNumber;
```

Beachten Sie, dass mit erster Schreibweise nur die erste Variable als Pointer deklariert wird, mögliche folgende sind einfache Variablen des Datentyps.

```
int* piNumber, a, b;
```

ist identisch zu

```
int a, b, *piNumber;
```

Hinweis: Mathematische Funktionen / Die Quadratwurzelfunktion sqrt

In der Standardbibliothek von C++ sind viele mathematische Funktionen schon implementiert. Um sie verwenden zu können, muss die Bibliothek `cmath` über folgenden Befehl eingebunden werden:

```
#include <cmath>
```

In `cmath` kann auch die Funktion `sqrt` zur Berechnung einer Quadratwurzel gefunden werden. Sie hat die folgende Signatur:

```
double sqrt (double x);
```

Eine weitere Funktion in `cmath` ist die Funktion `pow` zur Berechnung von Potenzen. Sie hat die folgende Signatur:

```
double pow (double dBase, double dExponent);
```

1.12 Referenzen

Die erste Neuerung von C++ gegenüber C lernen Sie nun kennen, die Referenz. Was eine Referenz ist, wie sie sich von Pointern unterscheidet und wo und wie man sie benutzt, erklären wir in den nächsten Abschnitten.

1.12.1 Definition

Eine Referenz ist eine Variable, deren Wert zu jeder Zeit dem Wert einer anderen Variablen entspricht. Änderungen am Wert dieser Variablen haben Änderungen am Wert der Referenz zur Folge. Umgekehrt wirken sich Änderungen am Wert der Referenz auch auf den Wert der referenzierten Variablen aus.

Die Referenz ist während ihrer Lebensdauer mit genau einer Variablen verbunden. Diese Variable muss bereits existieren, wenn die Referenz initialisiert wird. Wird die Variable gelöscht, so kann man auch auf die Referenz nicht mehr zugreifen.

1.12.2 Gründe für den Einsatz

Die Referenz wurde für C++ entwickelt, um viele Nachteile von Kopien und Pointern aufzuheben.

Der Vorteil von Referenzen gegenüber Kopien ist, dass man stets auf den aktuellen Wert der referenzierten Variablen zurückgreifen kann und der Wert deshalb nicht veraltet ist.

Der Vorteil gegenüber Pointern besteht im Sicherheitsgewinn. Erstens können Pointer auch noch existieren, wenn die referenzierte Variable gelöscht ist. Greift man dann auf den Pointer zu, dereferenziert man undefinierten Speicherbereich, was unweigerlich zu einem Fehler führt.

Zweitens kann man Referenzen im Normalfall nicht „umbiegen“, was die damit verbundenen Manipulationsmöglichkeiten verhindert.

Ein weiterer Vorteil gegenüber Pointern liegt in der vereinfachten Parameterübergabe beim Aufruf von Funktionen.

1.12.3 Syntax

Im Folgenden erhalten Sie eine kurze Einführung in die Syntax der Referenz.

1.12.3.1 Initialisierung

Wie in der Einleitung beschrieben, ist eine Referenz während ihrer Lebensdauer mit der referenzierten Variablen verbunden. Aus diesem Grund muss die Variable schon bestehen, wenn die Referenz erstellt wird. Referenzen können daher nur initialisiert werden. Eine Deklaration ist nicht möglich, weil die Referenz dann einen undefinierten Bezugswert hätte.

Hinweis: Deklaration, Definition und Initialisierung

Zur Erinnerung erklären wir noch einmal Deklaration, Definition und Initialisierung:

Deklaration: Macht dem Compiler Datentyp, Bezeichner und Dimension einer Variablen bekannt, um Speicher für die Variable anzulegen. Beispiel:

```
int iPieceNumber;
```

Definition: Zuweisung eines Wertes zu einer bereits deklarierten Variablen. Beispiel:

```
iPieceNumber = 35;
```

Initialisierung: Erstmalige Zuweisung eines Werts an eine Variable. Deklaration und Initialisierung in einem Beispiel:

```
int iPieceNumber = 35;
```

Eine Referenz initialisieren Sie durch Angabe des Datentyps der Bezugsvariable gefolgt von einem `&`-Operator, dem Bezeichner und der Zuweisung. Vergleichen Sie hierzu Listing 20.

Der `&`-Operator sollte unmittelbar hinter dem Datentyp stehen, da er andernfalls eine andere Bedeutung impliziert.

```
1  // Initialisierung einer Variablen
2  int iStudentID = 27002;
3
4  // Initialisierung auf die Variable
5  int& riStudentID = iStudentID;
```

Listing 20: Referenzinitialisierung

1.12.4 Zugriff über Referenzen

Referenzen können im Code genauso behandelt werden, wie die Variablen, die sie referenzieren.

1.12.4.1 Parameterübergabe per Referenz

Ihren großen Vorteil spielen Referenzen bei der Parameterübergabe aus. Sie sind einfacher zu handhaben als Pointer. Die Parameterübergabe läuft folgendermaßen ab:

1. Man erstellt eine Funktion, die eine Referenz als Übergabeparameter erwartet.
2. Man ruft die Funktion auf und übergibt ihr eine Variable.
3. Die Funktion bildet eine Referenz auf die übergegebene Variable und arbeitet damit.
4. Alle Änderungen, die die Funktion an der Referenz vornimmt, finden automatisch auch an der Variable statt.

Diese Schritte werden an einem Beispiel in Listing 21 gezeigt.

```
1   // Definition einer Funktion die eine Referenz und eine Variable als
2   // Uebergabeparameter erwartet
3   void changeStudentID(int& riOldStudentID, int iNewStudentID)
4   {
5      riOldStudentID = iNewStudentID;
6   }
7
8   // Die Funktion wird wie folgt in der Main aufgerufen
9   int main()
10  {
11     int iStudentIDDetlefCruse = 99933;
12     changeStudentID(iStudentIDDetlefCruse, 999773);
13     return 0;
14  }
```

Listing 21: Parameterübergabe per Referenz

Aufgabe: Referenzen

Implementieren Sie eine Funktion mit folgender Signatur:

`void swap (<Übergabeparameter 1>, <Übergabeparameter 2>).`

Diese Funktion soll die Werte der übergebenen Parameter tauschen. Verwenden Sie für die Implementierung Referenzen. Testen Sie die Funktion, indem Sie in einer main-Funktion zwei `int`-Variablen initialisieren und deren Wert anschließend tauschen.

1.13 Speicherverwaltung

In diesem Kapitel lernen Sie die beiden Arten der Speicherverwaltung in C++ kennen.

1.13.1 Statische Speicherverwaltung

Alle Variablen und Funktionen, die Sie im Quelltext nicht über Pointer definieren, werden statisch angelegt.

Der Arbeitsspeicherbereich, in dem sie gespeichert werden, heißt Stapelspeicher (Stack). Die Besonderheit des Stapelspeichers liegt darin, dass Elemente nur in umgekehrter Reihenfolge gelesen werden können, in der sie gespeichert wurden. Daher hat der Stapelspeicher auch seinen Namen. Die zuerst gespeicherten Elemente liegen ganz unten und nur das oberste Element kann abgenommen werden. Dies wird auch Last-In-First-Out-Prinzip (LIFO) genannt.

1.13.2 Dynamische Speicherverwaltung

Sie können Variablen und Funktionen im Quelltext auch dynamisch anlegen.

1.13.2.1 Definition

Nicht immer ist vor der Laufzeit eines Programms bekannt, wie viel Speicher für die Programmausführung notwendig ist. Beispielsweise kann man bei einem Texteditor nicht vorhersagen, wie groß der vom Bediener eingegebene Text wird. Ein weiteres Beispiel ist ein Datenbanksystem, bei dem zu Beginn keine Aussagen über die Datenmengen getroffen werden können.

Diesem Problem begegnet man, indem man zur Laufzeit Speicher reserviert. Dieser Vorgang wird als dynamische Speicherreservierung bezeichnet und wird mit den folgenden Operatoren umgesetzt.

1.13.2.2 `new`-Operator

Mit `new` können Sie zur Programmlaufzeit beliebige Variablen anlegen. Sie übergeben dem Operator den Datentyp der zu erzeugenden Variablen. Der Operator liefert dann einen Pointer auf genau eine solche, dynamisch im Speicher erzeugte Variable zurück.

An dem kurzen Beispiel in Listing 22 wollen wir Ihnen nun zeigen, wie Sie den `new`-Operator verwenden.

```
1  // Dynamische Speicherreservierung für eine Variable vom Typ double
2  double* pdWidth = new double;
3
4  // Dynamische Speicherreservierung für ein Array der Dimension 80 vom Typ char
5  char* pchName = new char[80];
```

Listing 22: Der `new`-Operator

1.13.2.3 `delete`-Operator

Werden die dynamisch erzeugten Variablen nicht mehr benötigt, müssen Sie sie explizit löschen. Auf diese Weise wird dem Laufzeitsystem dieser Speicherbereich als frei gemeldet und kann anderweitig verwendet werden.

Löscht man die Speicherbereiche nicht nach der Verwendung, so wächst der Speicherbedarf stetig an und das Programm kann sich verlangsamen.

Der zugehörige Operator zum Löschen ist `delete`, siehe Listing 23.

```
1  // Freigeben einer dynamisch angelegten Variable
2  delete pdWidth;
3
4  // Freigeben eines dynamisch angelegten Arrays
5  delete[] pchName;
```

Listing 23: Der delete-Operator

Hinweis: Freigeben eines Arrays

Um ein dynamisch reserviertes Array freizugeben, müssen sie *delete[]* verwenden. Verwenden sie nur *delete*, wird lediglich der Kopf des Arrays freigegeben.

Aufgabe: Speicher und Arrays

Manchmal werden Arrays mit sehr großen Dimensionen benötigt, z.B. bei einigen wissenschaftlichen Algorithmen. Der folgende Code zeigt eine Möglichkeit, ein solches Array anzulegen.

```
int main()
{
      int iStack[100000000];
      //Algorithmus…
      return 0;
}
```

a) Was würde bei der Ausführung des Programms geschehen? Wie ließe sich das Problem beheben?

b) Schreiben Sie ein Programm, das ein Array mit 100000 Elementen auf dem dynamischen Speicher anlegt und diese mit Zufallszahlen von 0 bis 100 füllt (siehe nachfolgender Hinweis). Anschließend soll ausgegeben werden, wie viele der erzeugten Zufallszahlen ohne Rest durch 13 teilbar sind. Denken Sie daran, den Speicherplatz des Arrays am Ende wieder freizugeben.

Beachten Sie die Hinweise.

Hinweis: Zufallszahlen

Die C++ Standardbibliothek stellt in der Bibliothek `<cstdlib>` die Funktion `rand` bereit, um (Pseudo-)Zufallszahlen zu erzeugen. Der Zufallsgenerator muss einmalig initialisiert werden (am besten am Anfang der `main()`-Funktion). Dies geschieht durch den Aufruf der Funktion `srand`, der ein Seed (Initalwert) übergeben wird. Damit bei jeder Ausführung des Programmes ein anderer Seed verwendet wird, wird über die Methode `time(nullptr)` aus der Bibliothek `<ctime>` die aktuelle Zeit übergeben.

Die Syntax für die Verwendung von `rand` lautet: `rand() % n + m`, was eine Zufallszahl zwischen `m` und `m+n` ergibt. Listing 24 zeigt die Verwendung von `rand` und `srand`.

```
1  // Einbinden der benötigten Bibliotheken
2  #include <cstdlib>
3  #include <ctime>
4
5  int main()
6  {
7     // Einmalige Initialisierung des random-seeds
8     srand(time(nullptr));
9
10    // Zufallszahl zwischen [m, m+n[, kann öfters verendet werden
11    int iNumber = rand() % n + m;
12
13    return 0;
14 }
```

Listing 24: Berechnung von Pseudo-Zufallszahlen

Aufgabe: Bubblesort

Bei der Arbeit mit Listen benötigt man des Öfteren Sortieralgorithmen. Informieren Sie sich im Internet über die Funktionsweise des Bubblesort-Algorithmus und

a) Implementieren Sie anschließend eine Funktion, die ein eindimensionales `int`-Array ordnet. Das Array sowie die -größe werden als Parameter übergeben.

b) Schreiben Sie eine main-Funktion, die ein `int`-Array der Größe 32 auf dem Heap (dynamische Speicherverwaltung) anlegt, mit Zufallszahlen füllt (siehe Hinweis oben) und mittels der in a) implementierten Funktion sortiert.

c) Machen Sie sich Gedanken über die Effizienz dieses Sortieralgorithmus.

Verständnisfragen

1. Was ist der Unterschied zwischen der dynamischen und statischen Speicherreservierung?
2. Wann benutzt man dynamische Speicherreservierung?
3. Welche Gefahren gehen von Pointern aus?
4. Welche Vorteile haben Referenzen gegenüber Pointern?

1.14 Idee der Objektorientierung

Um zu verstehen, was man unter dem objektorientierten Programmierparadigma versteht, wollen wir zunächst klären, was ein Programmierparadigma ist. Ein Programmierparadigma beschreibt ein übergeordnetes Denkmuster und legt damit die Struktur und den Ablauf eines Programmes fest.

Aus dem Grundstudium kennen sie bereits das imperative (d.h. befehlsorientierte) Programmierparadigma. Funktionen werden nacheinander aufgerufen und abgearbeitet. In C++ können sie auch weiterhin in diesem Programmierparadigma programmieren. Hinzu kommt jetzt noch die Objektorientierung.

Bisher bestanden Ihre Programme aus einer Vielzahl von Funktion und lokalen sowie globalen Variablen. Dabei kann jede Funktion und jede globale Variable an beliebiger Stelle im Programm aufgerufen bzw. verändert werden.

Aus dieser Struktur ergeben sich, insbesondere bei großen Projekten, zwei wesentliche Nachteile.

- Da globale Variablen überall zugänglich sind, können sie leicht manipuliert werden.
- Da die Funktionen als unstrukturierte Liste vorliegen, geht mit zunehmender Größe des Projektes die Übersicht verloren.

In den 80er Jahren stieß diese Art der Programmierung aufgrund dieser beiden Nachteile auf ihre Grenzen. Die immer weiter angewachsenen Softwareprojekte konnten nicht mehr gehandhabt werden. Das führte zur Entwicklung eines neuen Programmierparadigmas, der Objektorientierung.

Hinweis: Objektorientierte Programmierung

Imperatives und objektorientiertes Programmierparadigma sind nicht als Gegensätze zu verstehen, sondern können in C++ gemeinsam in einem Projekt verwendet werden. In Sprachen wie Java oder C# ist dies nicht der Fall.

Die grundlegende Neuerung der Objektorientierung liegt darin, dass Funktionen und Variablen in Klassen gruppiert werden.

1.15 Objekte

Ein Objekt ist eine Datenstruktur mit Bezeichner, die sowohl Variablen als auch Funktionen enthält. In diesem Kontext nennt man Variablen Attribute und Funktionen Methoden.

Bezeichner ist der eindeutige Name des Objektes.

Attribute enthalten Daten, die sich auf das zugehörige Objekt beziehen.

Methoden eines Objektes manipulieren die Attribute dieses Objekts.

Am Beispiel des Spiels „Ballerburg“ lernen Sie nun die Unterschiede in der Strukturierung eines Programms mit und ohne Objektorientierung kennen. Abbildung 11 zeigt die Oberfläche des Spiels.

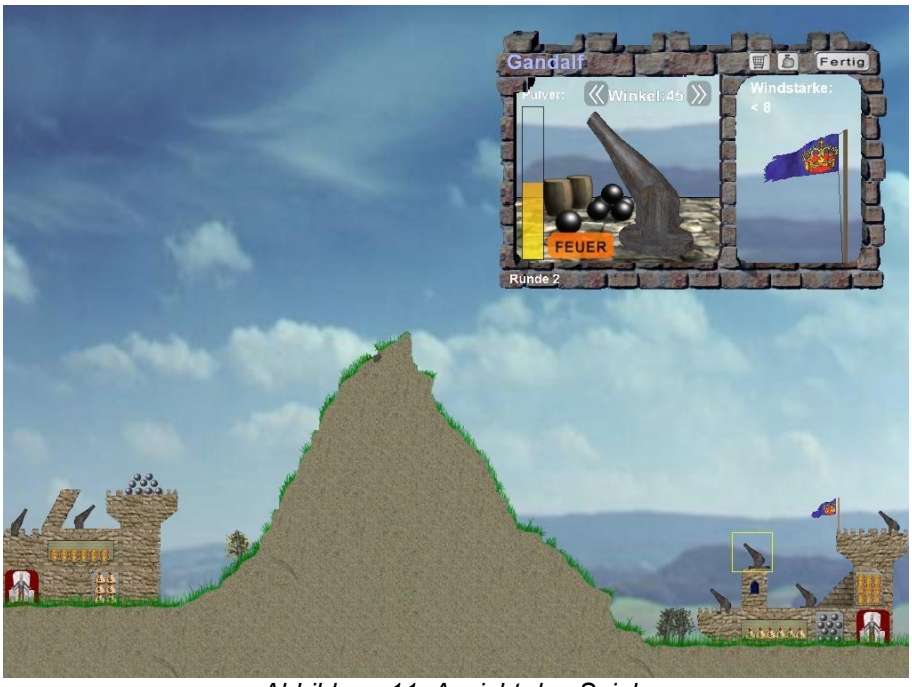


Abbildung 11: Ansicht des Spiels

Abbildung 12 deutet an, wie das Programm ohne die Objektorientierung implementiert sein könnte.

In Abbildung 13 sind die Funktionen und Variablen in Klassen organisiert.

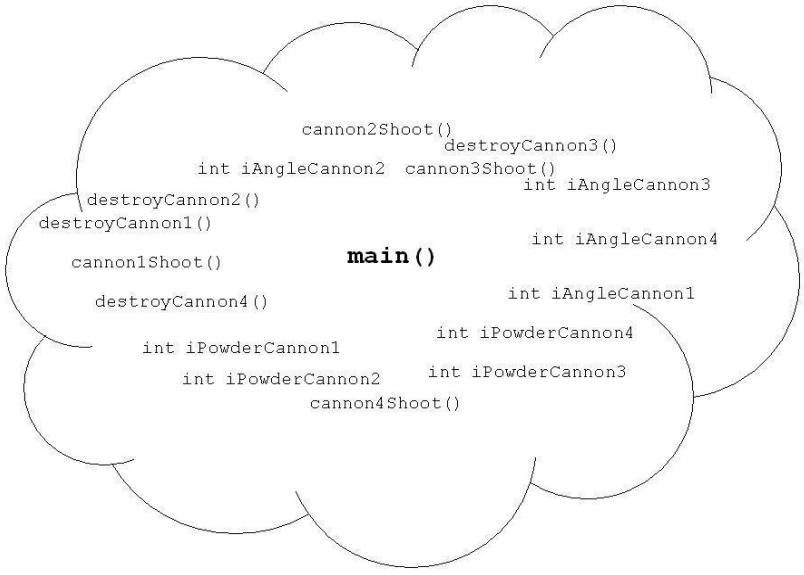


Abbildung 12: Durcheinander von Funktionen ohne Objektorientierung

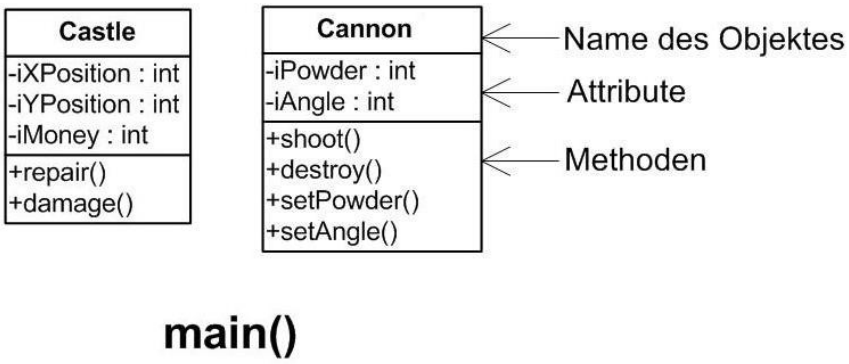


Abbildung 13: Strukturierung des Programms durch Objektorientierung

Verständnisfragen

- Was ist ein Objekt?
- Woraus besteht es?
- Welche möglichen Objekte fallen ihnen ein?

Um Objekte erstellen zu können, ist das Wissen von Klassen notwendig, da in der Objektorientierung jedes Objekt zu einer Klasse gehören muss. Aus diesem Grund machen wir sie im nachfolgenden Abschnitt mit Klassen vertraut.

1.16 Klassen

In diesem Kapitel stellen wir Ihnen den Begriff der Klasse vor und beschäftigen uns ausführlich mit ihrer Programmierung.

1.16.1 Zusammenhang zwischen Klassen und Objekten

Jedes Objekt gehört genau einer Klasse an. Die Objektidentität besagt, dass keine zwei Objekte identisch sind, d. h. jedes Objekt kann eindeutig identifiziert werden. Mithilfe einer Klasse lassen sich beliebig viele Objekte erstellen. Diese Objekte haben alle den gleichen Aufbau.

- Anzahl der Attribute,
- Datentypen der Attribute,
- Namen der Attribute,
- Standardwerte der Attribute,
- Zugriffsbeschränkungen auf die Attribute;
- Anzahl der Methoden,
- Deklaration der Methoden,
- Implementierung der Methoden,
- Zugriffsbeschränkungen auf die Methoden.

Hinweis: Merksatz

Die Klasse dient als Bauplan der zugehörigen Objekte.

Verständnisfragen

- Wie hängen Klasse und Objekt zusammen?
- Wie viele Objekte lassen sich aus einer Klasse erstellen?
- Können Objekte andere Objekte enthalten?

Wie man einen solchen Bauplan erstellt, erfahren Sie im nächsten Abschnitt.

1.16.2 Implementierung einer Klasse

Die Implementierung einer Klasse teilt sich in C++ in zwei Dateien auf, eine Header- und eine `cpp`-Datei. Die Dateien können beliebig benannt werden. Es bietet sich jedoch an und ist gängige Programmierpraxis, pro Klasse eine Header- und eine Implementierungsdatei zu verwenden und diese mit dem Klassennamen zu bezeichnen.

Die Headerdatei enthält sämtliche Deklarationen, d.h. in dieser Datei wird festgelegt:

- Anzahl der Attribute,
- Datentypen der Attribute,
- Namen der Attribute,
- Standardwerte der Attribute,
- Zugriffsbeschränkungen auf die Attribute;
- Anzahl der Methoden,
- Deklaration der Methoden,
- Zugriffsbeschränkungen auf die Methoden.

Dem Codebeispiel in Listing 25 können Sie den Aufbau einer solchen Headerdatei entnehmen, in Listing 25 die entsprechende `cpp`-Datei. Dabei verbleibt die main-Funktion in einer eigenen Datei, die gewöhnlich `main.cpp` genannt wird. Ein Projekt aus der Automatisierungstechnik könnte daher wie folgt aufgebaut sein:

- `Zylinder.h`
- `Zylinder.cpp`
- `Sensor.h`
- `Sensor.cpp`
- `Steuerung.h`
- `Steuerung.cpp`
- `main.cpp`

```
1   // Mehrfacheinbindung verhindern
2   #ifndef MONITOR_H
3   #define MONITOR_H
4
5   #include <string>
6
7   // Deklaration der Klasse
8   class Monitor
9   {
10  // Deklaration der Attribute
11  private:
12      std::string m_sManufacturer;
13      std::string m_sModel;
14      float m_fMaxHeight;
15      int m_iSize;
16  // Deklaration der Methoden
17  public:
18      Monitor();
19      ~Monitor();
20      void showMenue();
21      void enterSleepMode(int iWaitingTime);
22  };
23
24  #endif
```

Listing 25: Deklaration der Klasse Monitor

1. Zu Anfang muss, wie bei allen Headerdateien, die Mehrfacheinbindung verhindert werden.
2. Danach binden Sie die benötigten Headerdateien ein.
3. Mit dem Schlüsselwort class leiten Sie die Definition einer Klasse ein. Der Bezeichner der Klasse steht direkt hinter dem Schlüsselwort. In diesem Fall Monitor.
4. Es erfolgt die Deklaration der Attribute. Die Bedeutung der Schlüsselwörter private und public werden in Abschnitt 1.16.4.1 erklärt. Wofür der ::-Operator benötigt wird, entnehmen Sie dem Abschnitt 1.16.3.
5. Jetzt werden die Methoden deklariert. Bei den ersten beiden Methoden handelt es sich um Konstruktor und Destruktor. Diese werden im Abschnitt 1.16.5 gesondert behandelt.
6. Vergessen sie nicht, die Klassendeklaration mit einem Semikolon (`;`) zu schließen.

Die **Implementierungsdatei** enthält lediglich noch die Implementierung der Methoden, siehe Listing 26. Standardmäßig sollten alle Attribute hier mit einem Standardwert definiert werden, da sonst noch zufällige Werte im Bereich des Speichers vorhanden sein und unvorhergesehenes Verhalten auslösen können.

```
1   #include "Monitor.h"
2
3   // Konstruktor
4   Monitor::Monitor()
5   {
6      // Definition der Attribute
7      m_sManufacturer = "HP";
8      m_sModel = "DX7400";
9      m_fMaxHeight = 40;
10     m_iSize = 17;
11  }
12
13  // Destruktor
14  Monitor::~Monitor()
15  { /* ... */ }
16
17  // Methodendefinition
18  void Monitor::showMenue()
19  { /* ... */ }
20
21  // Methodendefinition
22  void Monitor::enterSleepMode(int iWaitingTime)
23  {
24     wait(iWaitingTime);
25     goToSleep();
26  }
```

Listing 26: Implementierung der Methoden der Klasse Monitor

1.16.3 Namensräume

Für C++ gibt es eine Vielzahl vorgefertigter Bibliotheken. Da sich die Entwickler nicht über die Namen der darin verwendeten Klassen, Methoden etc. abstimmen, kann es passieren, dass zwei Klassen, Methoden etc. aus unterschiedlichen Bibliotheken, denselben Namen haben. Das gleiche Problem kann auch bei großen selbstentwickelten Softwareprojekten auftreten. Die Benutzung eines solchen mehrfach definierten Bezeichners ruft einen Fehler im Programm hervor. Um dieses Problem zu beseitigen, ist man dazu übergegangen, den Inhalt jeder Bibliothek einem eigenen Namensraum zuzuordnen. I.d.R. werden die Bezeichner der Namensräume an den Firmennamen, Modulnamen oder Produktnamen orientiert.

Jede Klasse legt nochmal einen eigenen Unternamensraum für ihre Attribute und Methoden an. Will man außerhalb der Klassendeklaration auf den Namensraum zugreifen, muss man diesen explizit angeben. Dies geschieht mit Hilfe des Bereichsoperators, siehe Abschnitt 1.16.3.2.

1.16.3.1 Schlüsselwort `namespace`

Um einen eigenen Namensraum zu definieren, gehen Sie wie im Codebeispiel in Listing 27 vor.

```
1  #ifndef VEHICLE_H
2  #define VEHICLE_H
3
4  // Definition des eigenen Namensraumes Vehicle
5  namespace Vehicle
6  {
7     class Car
8     {
9     public:
10    Car();
11    ~Car();
12    };
13
14    class Bike
15    {
16    public:
17    Bike();
18    ~Bike();
19    };
20 }
21 #endif
```

Listing 27: Definition eines eigenen Namensraums

Welche Möglichkeiten es gibt, auf die Bezeichner eines Namensraumes zuzugreifen, zeigen die nächsten beiden Abschnitte.

1.16.3.2 Bereichsoperator ::

Um auf einen Bezeichner eines Namensraumes zuzugreifen, bedient man sich folgender Syntax.

```
<Namensraum>::<Bezeichner innerhalb dieses Namensraumes>
```

Codebeispiel Listing 28 soll Ihnen die Verwendung des std-Namensraums und des Bereichsoperators verdeutlichen.

```
1  #include <iostream>
2
3  int main()
4  {
5     int iNumber = 0;
6
7     std::cout << "Bitte geben sie eine natuerliche Zahl ein" << std::endl;
8     std::cin >> iNumber;
9     std::cout << "Sie haben eingegeben: " << iNumber << std::endl;
10
11    return 0;
12 }
```

Listing 28: Verwendung des Bereichsoperators

1.16.3.3 `using namespace` Direktive

Man kann einen Namensraum auch komplett einbinden. Schreibt man die Anweisung `using namespace <Namensraum>;` am Anfang einer Header- oder Implementierungsdatei, geht der Compiler bei unbekannten Bezeichnern davon aus,

dass diese aus dem angegebenen Namensraum stammen. **Dieses Konstrukt sollte jedoch so selten wie möglich eingesetzt werden, da sonst der Nutzen von Namensräumen ausgeschaltet wird.** In Listing 29 wird die Verwendung der Direktive `using namespace` demonstriert.

```
1   #include <iostream>
2
3   using namespace std;
4
5   void main ()
6   {
7      int iNumber = 0;
8
9      cout << "Bitte geben sie eine natuerliche Zahl ein" << endl;
10     cin >> iNumber;
11     cout << "Sie haben eingegeben: " << iNumber << endl;
12  }
```

Listing 29: Verwendung der `using namespace` Direktive

1.16.4 Kapselungsprinzip

Das Kapselungsprinzip stellt ein elementares Prinzip der Objektorientierung dar. Es ist wichtig zu verstehen, warum es angewandt werden muss und wie man es umsetzt.

1.16.4.1 Zugriffsspezifizierer

In diesem Abschnitt erklären wir Ihnen, was Zugriffsspezifizierer sind, welche es gibt und wie man diese anwendet.

Zugriffsspezifizierer sind die Schlüsselwörter `public`, `private` und `protected`. Sie erlauben es für jede Methode und jedes Attribut anzugeben, wie darauf zugegriffen werden darf.

`public` Das Attribut oder die Methode kann in jeder anderen Methode, auch außerhalb des Objekts aufgerufen werden. Dieser Aufruf geschieht durch:

`<Objektname>.<Attributbezeichnung>.`

Als Beispiel: `Printer.goToSleep(60);`

`private` Das Attribut/die Methode kann nur innerhalb der Methoden der eigenen Klassen aufgerufen werden. Der Aufruf erfolgt genauso wie oben. Eine Verwendung ist jedoch innerhalb der eigenen Methoden möglich.

`protected` Dieser Spezifizierer wird im Abschnitt Vererbung besprochen. Siehe Abschnitt 1.17.1

Wie eingangs bereits beschrieben, stellt die Manipulierbarkeit von Daten an jeder beliebigen Stelle im Programm ein Problem dar. Deshalb hat sich in der Objektorientierung das Prinzip der Kapselung etabliert. Kapselung bedeutet, dass die Attribute eines Objektes nur von ihm selbst und keinesfalls von außen geändert werden dürfen. D.h. alle Attribute müssen als `private` deklariert sein. Dadurch sind Aufrufe der Form `<Objektname>.<Attributbezeichnung> = …` nicht mehr möglich. Will man dennoch auf die Attribute zugreifen bedient man sich sogenannter `get`/`set`-Methoden, die man für jede Klasse implementieren muss.

1.16.4.2 `get()`- und `set()`–Methoden

Mithilfe der `get()`-Methoden kann man Attributwerte auslesen und mit den set()-Methoden Attributwerte manipulieren.

Der wesentliche Unterschied der `get()`-Methode zur direkten Ausgabe kann beispielsweise darin liegen, dass die Ausgabe formatiert wird.

Der wesentliche Unterschied der `set()`-Methoden zur direkten Zuweisung liegt darin, dass die Werte nicht ohne Überprüfung übernommen werden, sondern von ihnen innerhalb der Methoden überprüft werden können. So kann beispielsweise die Zuweisung einer negativen Zahl zu einem Attribut, welches das Alter angibt, verhindert werden.

Das Codebeispiel in Listing 30 zeigt am Beispiel der Klasse `Monitor` wie man `get()`- und `set()`-Methoden implementiert.

Verständnisfragen

- Was versteht man unter dem Kapselungsprinzip?
- Wie wird es realisiert?
- Was bewirken die Zugriffsspezifizierer `private` und `public`?
- Warum greift man nicht direkt auf Attribute zu, sondern verwendet `get()`– und set()-Methoden?

```cpp
// Monitor.cpp
#include "Monitor.h"

// Konstruktor
Monitor::Monitor()
{
   // Definition der Attribute
   m_sManufacturer= "HP";
   m_sModel        = "DX7400";
   m_fMaxHeight    = 40;
   m_iSize  = 17;
}
// Destruktor
Monitor::~Monitor()
{ /* ... */ }

void Monitor::setManufacturer(string sManufacturer)
{  // Die Klasse string enthält die Methode size welche die Größe
   // des jeweiligen String-Objektes zurückgibt.
   if(sManufacturer.size() <= 80)
   m_sManufacturer = sManufacturer;
}

void Monitor::setMaxHeight(float fMaxHeight)
{
   if(fMaxHeight > 0)
   m_fMaxHeight = fMaxHeight;
}

int Monitor::getSize()
{
   return m_iSize;
}

float Monitor::getMaxHeight()
{
   return m_fMaxHeight;
}

void Monitor::showMenu()
 {  /* ... */ }

void Monitor::enterSleepMode(int iWaitingTime)
{
   wait(iWaitingTime);
   goToSleep();
}
```

Listing 30: Implementierung von get und set Methoden

1.16.5 Konstruktor und Destruktor

Mit der Implementierung der Klasse legen wir den Bauplan für die daraus abgeleiteten Objekte fest. Um Objekte mit Hilfe dieses Bauplans zu erstellen, bedarf es einer besonderen Methode, des sogenannten Konstruktors.

Konstruktoren sind spezielle Methoden einer Klasse, die nur ein einziges Mal beim Erzeugen eines Objektes aufgerufen werden. Sie haben den Zweck Speicher für das neue Objekt zu reservieren und die Attribute gegebenenfalls zu initialisieren.

Folgende grundsätzliche Regeln gelten für Konstruktoren:

- Konstruktoren haben denselben Namen wie die Klasse.
- Konstruktoren haben keinen Rückgabedatentyp, auch nicht `void`.
- Man kann keine Pointer auf Konstruktoren definieren.

Die Verwendung der Konstruktoren erläutern wir in den folgenden Abschnitten.

1.16.5.1 Standardkonstruktor

Für Klassen, die selbst keinen Standardkonstruktor definieren, erzeugt der Compiler einen Standardkonstruktor.

Wie man einen Standardkonstruktor deklariert, definiert und aufruft zeigen die Codebeispiele in Listing 31, Listing 32 und Listing 33.

```
1   // PhotoAlbum.h
2   class PhotoAlbum
3    {
4    private:
5       int m_iNumberOfPhotos;
6       std::string m_sTitle;
7
8    public:
9       // Deklaration des Standardkonstruktors
10      PhotoAlbum();
11  };
```

Listing 31: Deklaration des Standardkonstruktors

```
1   // PhotoAlbum.cpp
2   // Standardkonstruktor
3   PhotoAlbum::PhotoAlbum()
4   {
5      // Beim Standardkonstruktor ist hier kein Code notwendig, jedoch eine Intialisierung der Attribute
6      // wünschenswert
7   }
```

Listing 32: Definition des Standardkonstruktors

```
1   #include "PhotoAlbum.h"
2
3   int main()
4   {
5      // Statische Erzeugung des Objektes MyHolidays der
6      // Klasse PhotoAlbum mit Standardkonstruktor
7      PhotoAlbum MyHolidays;
8
9      // Dynamische Erzeugung des Objektes Birthday der
10     // Klasse PhotoAlbum mit Standardkonstruktor
11     PhotoAlbum* pBirthdayPhotos = new PhotoAlbum;
12  }
```

Listing 33: Aufruf des Standardkonstruktors; Erzeugung eines neuen Objektes

Hinweis: Initialisierung

Im Normalfall sollen alle Attribute einer Klasse im **Konstruktor** auf einen **Standardwert** gesetzt werden, da es sonst zu schwerwiegenden Fehlern kommen kann.

1.16.5.2 Überladener Konstruktor

Manchmal ist es notwendig Objekte mit unterschiedlichen Attributwerten zu initialisieren. Dies ist dadurch gelöst, dass mehrere Konstruktoren für ein und dieselbe Klasse gleichzeitig existieren können. Sie unterscheiden sich lediglich durch die Übergabeparameter. Man spricht von der sogenannten Überladung. Mehr zum Thema Überladung finden sie im Abschnitt 1.16.8. Ein überladener Konstruktor kann Zuweisungen von Standardwerten zu Attributen beinhalten aber auch andere Operationen. Dies soll im Codebeispiel Listing 34 gezeigt werden.

```
1   // Standardkonstruktor
2   PhotoAlbum::PhotoAlbum()
3   {
4      m_iNumberOfPhotos = 0;
5      m_sTitle = "My Photo Album";
6   }
7
8   // Überladener Konstruktor
9   PhotoAlbum::PhotoAlbum(int iNumberOfPhotos, std::string sTitle)
10  {
11     m_iNumberOfPhotos = iNumberOfPhotos;
12     m_sTitle = sTitle;
13  }
```

Listing 34: Definition eines Standardkonstruktors und eines überladenen Konstruktors

Die Parameterliste kann beispielsweise Attributwerte enthalten, die innerhalb des Konstruktors definiert werden. Die Deklaration eines solchen überladenen Konstruktors sehen Sie im Codebeispiel Listing 35.

```
1   // PhotoAlbum.h
2   class PhotoAlbum
3   {
4   private:
5       int m_iNumberOfPhotos;
6       std:string m_sTitle;
7
8   public:
9       // Deklaration des Standardkonstruktors oder überladener Konstruktors
10      // ohne Parameterliste
11      PhotoAlbum();
12
13      // Deklaration des überladenen Konstruktors mit Parameterliste
14      PhotoAlbum(int iNumberOfPhotos, std::string sTitle);
15  };
```

Listing 35: Definition eines überladenen Konstruktors mit Parameterliste

Konstante Variablen und Referenzen können bekanntlich nicht definiert werden, sondern müssen gleich initialisiert werden. Initialisierungen sind mit überladenen oder Standard-Konstruktoren alleine nicht möglich. Um Initialisierungen vornehmen zu können, muss man den Konstruktor deshalb um eine Initialisierungsliste ergänzen.

1.16.5.3 Initialisierungsliste

Um die Attribute gleich bei der Erzeugung des Objekts zu initialisieren, ergänzt man die Konstruktor-Definition um eine Initialisierungsliste. Dies ist zum einen notwendig bei der Vererbung (siehe 1.17) oder falls ein Objekt nicht durch den Standardkonstruktor initialisiert werden soll.

Die Syntax der Initialisierungsliste folgt folgenden Regeln:

- Die Initialisierungsliste beginnt direkt hinter dem Konstruktor mit einem Doppelpunkt
- Darauf folgt der Name des ersten Attributes.
- Hinter jedem Attribut steht in runden Klammern der Initialisierungswert.
- Folgen weitere Attributinitialisierungen, so werden diese durch Komma getrennt.
- Erst nach der Initialisierungsliste steht der Codeblock.

Das Codebeispiel in Listing 36 zeigt die Verwendung der Initialisierungsliste.

```
1   // Standard Konstruktor mit Initialisierungsliste
2   PhotoAlbum::PhotoAlbum() : m_iNumberOfPhotos(0), m_sTitle("My Photo Album")
3   { /* ... */ }
4
5   // Überladener Konstruktor mit Initialisierungsliste
6   PhotoAlbum::PhotoAlbum(int iNumberOfPhotos, std::string sTitle) :
7       m_iNumberOfPhotos(iNumberOfPhotos), m_sTitle(sTitle)
8   { /* ... */ }
```

Listing 36: Definition von Konstruktoren mit Initialisierungslisten

Hinweis: Initialisierungsliste

Die Initialisierungsliste ist darüber hinaus effizienter als ein normaler Konstruktoraufruf ohne Initialisierungsaufruf. Dies liegt daran, dass anstatt der zwei Schritte, Deklaration und Definition, nur noch ein Schritt, die Initialisierung, nötig ist.

1.16.5.4 Destruktor

Wie der Name bereits vermuten lässt, handelt es sich beim Destruktor um das Gegenstück des Konstruktors. Während der Konstruktor die Objekte erschafft, zerstört der Destruktor sie wieder. Er gibt den von den Objekten belegten Speicher wieder frei.

Für den Destruktor gelten folgende Regeln:

- Der Name des Destruktors ist ~<Klassenname> z.B. ~PhotoAlbum.
- Genau wie der Konstruktor besitzt er keinen Rückgabeparameter, auch nicht `void`.
- Jede Klasse besitzt nur einen einzigen Destruktor.
- In der Regel erfolgt der Aufruf des Destruktors implizit durch den Compiler bei Beendigung des Programmes.

Meist ist es ausreichend den Standarddestruktor zu verwenden, d.h. den Implementierungsblock wie beim Standardkonstruktor leer zu lassen. Wurde jedoch im Konstruktor dynamisch Speicher angefordert, muss dieser im Destruktor auch wieder manuell freigegeben werden.

Im Codebeispiel in Listing 37 finden Sie ein Beispiel für einen Destruktor.

```
1   // Standardkonstruktor der dynamisch Speicher anfordert
2   MP3Player::MP3Player()
3   {
4       m_psPlaylistElement = new string[5];
5   }
6
7   // Destruktor der den dynamisch angeforderten Speicher wieder frei gibt
8   MP3Player::~MP3Player()
9   {
10      delete[] m_psPlaylistElement;
11  }
```

Listing 37: Definition eines Destruktors

Verständnisfragen

- Welche Aufgabe hat der Konstruktor?
- Welche Aufgabe hat der Destruktor?
- Weshalb verwendet man überladene Konstruktoren?
- Wann muss man Initialisierungslisten verwenden?

1.16.6 Zugriff auf Methoden und Attribute

Es gibt zwei Arten des Zugriffs, diese sollen nachfolgend gezeigt werden.

1.16.6.1 Zugriff innerhalb der Klassendefinition

Von einem Zugriff innerhalb der Klassendefinition spricht man, wenn eine Klasse innerhalb einer Methodendefinition auf ihre eigenen Attribute und Methoden zugreift.

Unabhängig ob Attribute und Methoden als `public`, `protected` oder `private` spezifiziert sind, kann die Klasse immer auf ihre eigenen Attribute und Methoden zugreifen.

Um innerhalb einer Methodendefinition einer Klasse auf ihre Attribute und Methoden zuzugreifen, benötigen Sie keinen speziellen Operator. Der Methoden- bzw. der Attributname reicht für den Zugriff aus. Siehe dazu auch Codebeispiel in **Fehler! V erweisquelle konnte nicht gefunden werden.**.

1.16.6.2 Zugriff von außen

Unter einem Zugriff von außen versteht man den Aufruf eines Attributes oder einer Methode außerhalb der Klassendefinition, d.h. in den Definitionen anderer Klassen oder Funktionen. Der Zugriff von außen ist durch die Zugriffsspezifizierer geregelt. Gemäß dem Kapselungsprinzip sind alle Attribute einer Klasse als `private` deklariert. Aus diesem Grund kann man von außen nicht direkt auf sie zugreifen. Auf Methoden kann man von außen zugreifen, falls sie `public` deklariert sind. Dafür wird der Punkt- bzw. Pfeil-Operator benötigt.

Hinweis: Pfeiloperator vs. Punktoperator

Pfeiloperator

Hat man einen Pointer auf ein Objekt und will man über diesen auf die Attribute und Methoden eines Objektes zugreifen, benötigt man den Pfeiloperator `->`. Der Aufruf sieht folgendermaßen aus:

```
pBirthdayPhotos->m_iNumberOfPhotos
```

Punktoperator

Spricht man das Objekt direkt oder über eine Referenz an, so benötigt man den Punktoperator.

```
<Objektname>.<Attribut/Methode>
```

Siehe dazu auch Codebeispiel **Fehler! Verweisquelle konnte nicht gefunden w erden.**.

```
1   // AirCondition.h
2
3   class AirCondition
4   {
5   private:
6       float m_fTemperature;
7       float m_fAirHumidity;
8
9   public:
10      // Konstruktor und Destruktor
11      AirCondition();
12      ~AirCondition();
13      // get und set Methoden
14      float getTemperature();
15      bool setTemperature(float fTemperature);
16      float getAirHumidity();
17      void displayInfo();
18  };
```

Listing 38: Deklaration der Klasse AirCondition

```
1   // AirCondition.cpp
2
3   #include "AirCondition.h"
4   #include <iostream>
5
6
7   // Standardkonstruktor mit Initialisierungsliste
8   AirCondition::AirCondition() :
9   m_fAirHumidity(73.4)
10  { /* ... */ }
11
12  // Standarddestruktor
13  AirCondition::~AirCondition()
14  { /* ... */ }
15
16  // Implementierung der get Methode
17  float AirCondition::getTemperature()
18  {
19      // Zugriff auf ein Attribut innerhalb der Klasse
20      return m_fTemperature;
21  }
22
23  // Implementierung der set Methode
24  bool AirCondition::setTemperature(float fTemperature)
25  {
26      if(fTemperature>16.0 && fTemperature<30.0)
27      {
28              m_fTemperature = fTemperature;
29              return true;
30      }
31      else
32      {
33              return false;
34      }
35  }
36  // … Weiter auf der nächsten Seite
```

```
37 //Implementierung der get Methode
38 float AirCondition::getAirHumidity()
39 {
40    return m_fAirHumidity;
41 }
42
43 //Implementierung einer Methode, welche auf andere Methoden der Klasse zugreift
44 void AirCondition::displayInfo()
45 {
46    std::cout << "Temperatur ist: " << getTemperature() <<" Grad C" << std::endl;
47    std::cout << "Luftfeuchtigkeit ist: " << getAirHumidity() << " %" << std::endl;
48 }
```

Listing 39: Definition der Klasse AirCondition

```
1  // main.cpp
2  #include "AirCondition.h"
3
4  int main()
5  {
6     // Erzeugen des Objektes durch Aufruf des Standardkonstruktors
7     AirCondition airConditionMensa;
8
9     // Aufruf einer Methode der Klasse AirConditon von außen mithilfe des Punkt Operators.
10    airConditionMensa.setTemperature(18.0);
11
12    airConditionMensa.displayInfo();
13    return 0;
14 }
```

Listing 40: Verwendung der Klasse in der main-Funktion

Aufgabe: Flaschen

a) Erstellen Sie die Klasse `Flasche` mit den folgenden Inhalten:

- Attribute: `dVolumen` (`double`), `sMaterial` (`string`)

b) Fügen Sie der Klasse `Flasche` einen Standardkonstruktor hinzu.

c) Fügen Sie für jedes Attribut eine `get`- sowie `set`-Methode hinzu.

d) Erstellen Sie die Methode `printFlasche()`, welches die Attribute in der Konsole ausgibt.

e) Implementieren Sie eine Methode `adoptFlasche(…)`, welcher eine Flasche2 übergeben wird und die Attribute der ursprünglichen Flasche1 denen der übergebenen Flasche2 angleicht.

f) Erstellen Sie eine main-Funktion, in welcher Sie zwei Objekte der Klasse `Flasche` erstellen. Testen Sie alle erstellten Methoden.

Aufgabe: Datum – Finden Sie die Fehler

in den folgenden Listings sind die Definition einer Klasse `Date` (`Date.h`), deren Implementierung (`Date.cpp`) und eine main-Funktion gegeben. Die `main`-Funktion (`main.cpp`) erstellt Objekte der Klasse `Date`.

Die Definition von `Date`, sowie die Funktion `main` können Fehler enthalten. Verschaffen Sie sich einen Überblick über die Klasse `Date`. Sehen Sie nun die Funktion `main` durch und bestimmen Sie, ob und welche Befehle nicht funktionieren. Finden Sie die Ursache der Fehler und korrigieren Sie die Fehler im Skript, **ohne** das Programm in Eclipse zu übernehmen und zu kompilieren.

```
1   // Date.h
2   #ifndef DATE_H
3   #define DATE_H
4
5   class Date
6   {
7   private:
8       int m_day, m_month, m_year;
9       Date(int day, int month, int year);
10
11  public:
12      Date(int day, int month, int year);
13      bool isEqual(Date dd);
14      int getDay();
15      int getMonth();
16      int getYear();
17  };
18
19  #endif
```

Listing 41: Definition der Klasse Date, Date.h

```
1   // Date.cpp
2   #include "Date.h"
3
4   Date::Date(int day, int month, int year)
5   {
6       m_day = day; m_month = month; m_year = year;
7   }
8
9   int getDay() {return m_day;}
10  int Date::getMonth() {return m_month;}
11  int Date::getYear() {return m_year;}
12  bool Date::isEqual(Date dd)
13  {
14      if(m_day==dd.m_day && m_month==dd.m_month && m_year==dd.m_year)
15              return true;
16      return false;
17  }
```

Listing 42: Date.cpp

```
1  // main.cpp
2  // Was bewirken die einzelnen Zeilen? Finden Sie Fehler!
3  int main()
4  {
5     Date d1 = Date(2,2,2011);
6     d1.m_day = 15;
7     Date d2 = Date(15,2,2011);
8     bool tf = d2.isequal(d1);
9  }
```

Listing 43: Verwendung der Klasse Date in main, main.cpp

Aufgabe: Erweiterte Datumsklasse

Die Klasse `Date` aus dem vorigen Beispiel ist noch unvollständig und unpraktisch. Sie soll nun verbessert werden. Verwenden Sie die korrigierte Klasse Date aus der vorigen Aufgabe.

a) Fügen Sie der Klasse einen Standardkonstruktor hinzu, der ein zufälliges Datum erstellt. Achten Sie auf die Gültigkeit des Datums. Schaltjahre werden nicht berücksichtigt, ein gültiges Jahr liegt zwischen 1970 und 2030. (siehe Hinweis zur Aufgabe „Speicher und Arrays“)

b) Fügen Sie der Klasse eine Funktion `compare` hinzu, die testet ob ein übergebenes Datum zeitlich vor oder nach dem aufrufenden Datumsobjekt liegt (Rückgabewert 1: vor, 0: gleich, -1: nach)

c) Testen Sie die Klasse und alle Funktionen in einer `main`-Funktion. Erstellen Sie hierzu mehrere Objekte vom Typ Datum, vergleichen Sie sie untereinander und lassen Sie die Objekte in der Konsole ausgeben.

1.16.7 Variablen static und const

Statische Inhalte einer Klasse sind in allen Instanzen der Klasse gleich. D.h. hat eine Klasse eine statische Methode oder Variable, so wird diese von allen Instanzen der Klasse „geteilt“. Außerdem können statische Inhalte auch verwendet werden, wenn keine Instanz der Klasse existiert. Statische Inhalte werden mit dem vorangestellten Schlüsselwort `static` definiert.

1.16.7.1 Statische Variablen

Statische Variablen werden wie auch andere Variablen in der Header-Datei einer Klasse definiert. Listing 44 zeigt wie dies durch Voranstellen des Schlüsselworts `static` geschieht.

```
1  // Animal.h
2  class Animal
3  {
4  public:
5     static int siAnz; // Deklaration einer statischen Variable
6  };
```

Listing 44: Deklaration einer statischen Variablen

Eine Besonderheit von statischen Variablen ist, dass sie vor ihrer Verwendung initialisiert werden müssen. Dies geschieht in der Cpp-Datei durch das Einfügen folgender Zeile:

```
1  // muss in Animal.cpp eingefügt werden
2  int Animal::siAnz = 0; // Initialisierung der statischen Membervariable
```

Listing 45: Initialisierung einer statischen Variablen

1.16.7.2 Statische Methoden

Statische Methoden können auch ohne einer Instanz der Klasse verwendet werden. Sind werden also von der Klasse verwendet und nie von einer einzelnen Instanz. Daher wird eine statische Methode auch nicht durch einen Punkt-Operator oder einen Pfeiloperator aufgerufen, sondern durch den Bereichsauflösungsoperator :: . Nicht-statische Inhalte können von statischen Methoden nicht verwendet werden, da sie unabhängig von den Instanzen funktionieren und keinen Zugriff auf deren Attribute besitzen.

Eine häufige Verwendung von statischen Variablen und Methoden ist die Identifikation von einzelnen Instanzen sowie das Zählen aller vorhandenen Instanzen einer Klasse. In Listing 46 und Listing 47 wird ein Beispiel dargestellt.

```
1  // Bike.h
2  class Bike
3  {
4  public:
5      Bike();
6      int m_ID;                    // ID der jeweiligen Instanz
7      static int getCount();       // Deklaration der statischen Methode
8  private:
9      static int siCountID;        // Deklaration der statischen Variable
10 };
```

Listing 46: Headerdatei mit statischen Mitgliedern

```
1  // Bike.cpp
2  int Bike:: siCountID = 0;       // Initialisierung der statischen Membervariable
3
4  Bike::Bike()
5  {
6      siCountID++;                 // Erhöhung des Counters
7      m_ID = siCountID;            // Zuweisung der ID zur neu erstellten Instanz
8  }
9
10 int Bike::igetCount()            // gibt die Anzahl der erstellten Instanzen zurück
11 {
12     return siCountID;
13 }
```

Listing 47: Cpp-Datei mit statischer Methode und Initialisierung einer statischen Variablen

Der Zugriff auf statische Methoden erfolgt nach dem Schema `Klassenname::Methodenname` wie in Listing 48 gezeigt.

```
1   #include <iostream>
2   #include “Bike.h”
3
4   void main()
5   {
6      std::cout << “Anzahl der Fahrraeder: “ << Bike::getCount();
7   }
```

Listing 48: Verwendung von statischen Methoden

1.16.7.3 Das Schlüsselwort const

Kennzeichnet Funktionen, die Member-Variablen nicht verändert. Bei Aufruf kann auf die Variablen nur lesend zugegriffen werden. Dies kann dem Programmierer dienen, ist aber in bestimmten Fällen vom Compiler gefordert, bspw. bei Referenzen. Soll beispielsweise eine `get`-Funktion mit Hilfe einer konstanten Referenz auf ein Objekt aufgerufen werden, so muss die aufgerufene Funktion ebenfalls als `const` gekennzeichnet sein (siehe Listing 49 und Listing 50).

```
1   // Flasche.h
2   class Flasche
3   {
4   public:
5      Flasche();
6      Flasche(const Flasche &flasche);          // Überladener Konstruktor mit konstanter Referenz
7                                                // auf ein Objekt der Klasse Flasche
8      double getVolumen() const;                // konstante get-Funktion
9   private:
10     double m_dVolumen;
11  };
```

Listing 49 Headerdatei mit Nutzung von const

```
1   // Flasche.cpp
2   #include “Flasche.h“
3   Flasche()
4   {
5      m_dVolumen = 1.0;
6   }
7
8   Flasche(const Flasche &flasche)
9   {
10     M_dVolumen = flasche.getVolumen();
11  }
12
13  double getVolumen() const
14  {
15     return m_dVolumen;
16  }
```

Listing 50 Sourcedatei mit Nutzung von const

Aufgabe: Fahrzeuge

a) Erstellen Sie eine Klasse `vehicle`, die ein Fahrzeug repräsentiert. Jedes Fahrzeug hat eine Farbe (Blau, Rot, Grün, Weiß, Schwarz). Benutzen Sie eine Enumeration (siehe folgender Hinweis), um die Farbe darzustellen.
Die Enumeration soll zu der Klasse `vehicle` gehören, das heißt sie wird in dieser definiert. Wichtig ist, dass die Enumeration als erstes Element der Klasse definiert wird, damit nachfolgende Methoden darauf zugreifen können und dass sie als öffentlich deklariert wird, damit sie auch von außerhalb der Klasse verwendet werden kann (z.B. für den Konstruktor).
Zusätzlich hat jedes Fahrzeug einen Preis und ein Baujahr. Außerdem ist jedes Fahrzeug durch eine automatisch generierte, eindeutige Nummer identifizierbar (vgl. Listing 47). Überlegen Sie sich angemessene Datentypen für die Eigenschaften. Alle Eigenschaften werden bei der Initialisierung übergeben und müssen abgefragt werden können. Die Farbe des Fahrzeugs soll als `string` ausgegeben werden.

b) Fügen Sie der Klasse eine statische get()-Funktion `isOldtimer(…)` hinzu, die für ein übergebenes Fahrzeug zurückgibt, ob es vor 1980 gebaut wurde. Testen Sie die Klasse in einer `main`()-Funktion.

Hinweis: Enumerationen

Eine Enumeration ist ein Datentyp, der ein vom Benutzer festgelegtes Set an Integer Werten aufnehmen kann. Eine Enumeration wird wie folgt deklariert:

```
enum type_name {value0, value1, value2, …} object_names;
```

Dies erstellt den Typ `type_name`, welcher die Werte `value0`, `value1`, usw. annehmen kann. Objekte (Variablen) vom Typ `type_name` können direkt als `object_names` instanziiert werden. Die Bezeichner `value0`, `value1`, usw. werden implizit in Integer Werte konvertiert. Wenn nicht anders angegeben, so wird `value0` zu 0 konvertiert, `value1` zu 1, usw. Das anschließende Listing 51 zeigt die Verwendung einer Enumeration.

```
1  // Deklaration einer Enumeration
2  // Die Bezeichner in color haben die Integer-Werte: red=5, orange=6, green=7
3  enum color{red=5, orange, green};
4
5  // Instanziierung eines Objektes
6  color trafficlight;
7
8  // Initialisierung des Objekts
9  trafficlight = orange;
10
11 if(trafficlight == color::orange)
12    trafficlight = color::red;
13 else
14    trafficlight = color::orange;
```

Listing 51: Verwendung von Enumerationen

Verständnisfragen

- Was versteht man unter einem Zugriff innerhalb der Klassendefinition?
- Was versteht man unter einem Zugriff von außen?
- Worin unterscheiden sich diese beiden Zugriffsarten?
- Wann benötigt man den Punkt- und wann den Pfeiloperator?

1.16.8 Überladung

1.16.8.1 Überladung von Methoden

Anders als in C ist es in C++ möglich, ähnlich wie beim Konstruktor, mehreren Methoden einer Klasse denselben Namen zu geben. Sie müssen sich nur in den Übergabeparametertypen unterscheiden. Diese Methoden mit gleichen Namen, aber unterschiedlicher Übergabeparameter, nennt man überladene Methoden.

Das folgende Codebeispiel in Listing 52 zeigt die Überladung von Methoden.

```
1   class Geometry
2   {
3   private:
4       // Hier stehen die Attribute der Klasse
5
6   public:
7       // Konstruktor und Destruktor der Klasse
8       Geometry();
9       ~Geometry();
10      // Überladene Methoden der Klasse die eine Kreisfläche aus dem Radius berechnet
11      float CalculateCircleArea(float fRadius);
12      double CalculateCircleArea(double dRadius);
13
14      // Weitere Methoden
15
16  };
```

Listing 52: Deklaration überladener Methoden

Wie Sie anhand des Beispiels sehr schön sehen können, ermöglicht Überladung einen zusätzlichen Bedienkomfort für den Entwickler. Möchte man in C eine Funktionalität mit unterschiedlicher Art und Anzahl von Parametern umsetzen, so benötigt man dafür mehrere Funktionen, die unterschiedliche Namen haben müssen. In C++ kann man nun auf die vielen unterschiedlichen Funktionsnamen verzichten und die Funktionen in überladenen Methoden mit demselben Namen zusammenfassen. Je nach Art und Anzahl der Übergabeparameter wird beim Aufruf nun statisch prüfbar die richtige Methode ausgewählt.

1.16.8.2 Überladung von Operatoren

Nicht nur Methoden, sondern auch Operatoren (wie +, << usw.) können in C++ überladen werden. Dies ermöglicht es dem Programmierer, die entsprechenden Operationen auch für selbst definierte Datentypen zur Verfügung zu stellen. Das Überladen von Operatoren funktioniert prinzipiell wie bei Methoden. Als Beispiel sei die Klasse `complex` gegeben, die es ermöglichen soll, mit komplexen Zahlen zu rechnen (Legende: - private; + public).

complex
- ReTeil: double

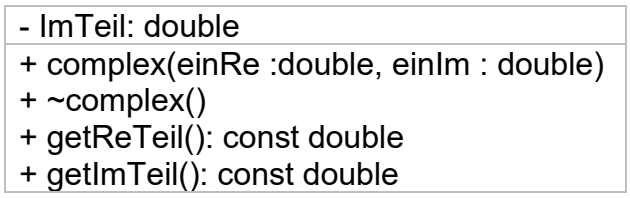


Abbildung 14: UML Klassendiagramm der Klasse complex

Im Folgenden sollen die Operatoren „+“ und „<<“ für die Klasse `complex` überladen werden. Die Operatoren gehören nicht zur Klasse `complex`, sondern in den globalen Namespace. Es empfiehlt sich daher, diese Operatoren in derselben Datei wie die `main`-Methode zu überladen. Damit die `main`-Methode, oder auch andere Methoden, die überladenen Operatoren verwenden können, müssen diese vor den entsprechenden Methoden deklariert werden.

Um komplexe Zahlen addieren zu können, muss der `+`-Operator überladen werden. Listing 53 zeigt, wie das geht.

```
1   complex operator+(complex a, complex b)
2   {
3       return complex(a.getReTeil() + b.getReTeil(), a.getImTeil() + b.getImTeil());
4   }
5
6   int main()
7   {
8       complex a = complex(2.1,2.0);
9       complex b = complex(4.0,3.3);
10      complex c = a+b;
11      return 0;
12  }
```

Listing 53: Überladung des +-Operators

Häufig ist es notwendig einen selbst definierten Datentyp in der Konsole (oder in einer Datei) auszugeben. Hierzu kann man den Shift-Operator `<<` überladen. Für die Klasse `complex` funktioniert das folgendermaßen:

```
1   #include <iostream>
2   #include <string>
3
4   // Klasse complex
5   std::ostream& operator<<(std::ostream &os, const complex &co)
6   {
7       return os << co.getReTeil() << " + " << co.getImTeil() << "i";
8   }
9
10  int main()
11  {
12      complex a = complex(2.0, 4.0);
13      std::cout << "a = " << a << std::endl;        // Ausgabe in Konsole: a = 2.0 + 4.0i
14      return 0;
15  }
```

Listing 54: Überladung des <<-Operators

Aufgabe: Quadrate

Es soll in mehreren Schritten eine Klasse geschrieben werden, die Objekte vom Typ Quadrat erstellt und die Addition, Subtraktion sowie die Ausgabe eines Quadrats in der Konsole ermöglicht. Ein Quadrat lässt sich durch die Kantenlänge repräsentieren.

a) Erstellen Sie eine Klasse `square`, die Quadrate repräsentiert. Ein Quadrat wird durch die Übergabe der Kantenläge an den Konstruktor erstellt und ist danach nicht mehr modifizierbar.

b) Von einem Quadrat können Länge, Fläche sowie Umfang abgefragt werden. Schreibe entsprechende Methoden, die diese Funktionalität zur Verfügung stellen.

c) Erstellen Sie einen zusätzlichen Konstruktor, dem eine konstante Referenz auf ein bereits existierendes Quadrat übergeben wird. Dieser erstellt dann ein neues Quadrat mit derselben Kantenlänge wie das übergebene Objekt.

d) Quadrate sollen addiert und subtrahiert werden können (hier durch Addition/Subtraktion der Kantenlängen). Überladen Sie hierzu den `+` und `-` Operator, sodass Objekte vom Typ `square` addiert und subtrahiert werden können (für die Syntax: siehe obiger Abschnitt).

e) Überladen Sie den Operator `<<`, sodass Objekte vom Typ `square` in der Konsole ausgegeben werden können. Die Ausgabe sollte in etwa so aussehen:

Quadrat: Kantenlänge=10, Fläche=100, Umfang=40.

Verständnisfragen

- Was versteht man unter der Überladung von Methoden und Operatoren?
- Wann wendet man die Überladung an?
- Wie überlädt man Methoden bzw. Operatoren?

1.17 Vererbung

Die Vererbung ist ein wichtiger Mechanismus des objektorientierten Programmierparadigmas.

1.17.1 Definition

Die Vererbung ist eine Beziehung zwischen zwei Klassen. Die eine Klasse nennt man Elternklasse, die andere Kindklasse. Die Kindklasse ist von der Elternklasse abgeleitet. Das heißt die Kindklasse enthält (erbt) alle Attribute und Methoden der Elternklasse. Zusätzlich können innerhalb der Kindklasse noch weitere Attribute und Methoden implementiert werden.

Auch wenn alle Attribute und Methoden der Elternklasse in der Kindklasse vorhanden sind, so ist der Zugriff der Kindklasse auf die vererbten Attribute und Methoden abhängig von den Zugriffsspezifizierern der Elternklasse.

`public` Alle Attribute und Methoden, die in der Elternklasse als `public` deklariert wurden, verhalten sich in der Kindklasse so, als wären sie dort als `public` deklariert. D.h. sowohl die Methoden der Kindklasse als auch fremde Objekte können die vererbten Methoden aufrufen.

`protected` Alle Attribute und Methoden, die in der Elternklasse als `protected` deklariert wurden, lassen sich nur innerhalb der Eltern- und Kindklasse aufrufen. Fremde Objekte haben darauf keinen Zugriff.

`private` Alle Attribute und Methoden, die in der Elternklasse als `private` deklariert wurden, können nur in der Elternklasse verwendet werden. Von außen sind sie weder sichtbar noch verwendbar. Gemäß dem Kapselungsprinzip (siehe Abschnitt 1.16.4) sind alle Attribute einer Klasse als `private` deklariert. Aus diesem Grund kann man in der Kindklasse nie direkt auf die von der Elternklasse vererbten Attribute zugreifen. Das Ein- und Auslesen der Attribute ist nur möglich, wenn `get` und `set` Methoden implementiert wurden.

Durch das Prinzip der Vererbung wird ermöglicht, dass Klassen mit ähnlichen Methoden und Attributen voneinander abgeleitet werden können, sie müssen somit nur einmal implementiert werden.

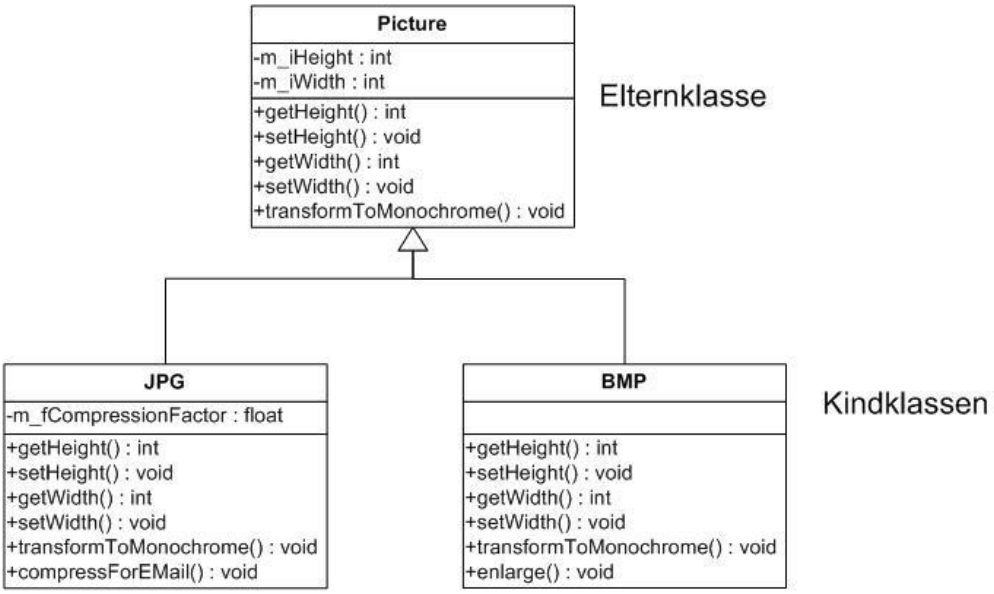


Abbildung 15: Beispiel einer Vererbung

Von jeder Klasse lassen sich beliebig viele Kindklassen ableiten. illustriert ist eine Vererbungsbeziehung.

1.17.2 Syntax

Wie Sie im Codebeispiel in Listing 55 sehen, müssen Elternklassen nicht besonders gekennzeichnet werden.

Die Zugehörigkeit einer Kindklasse zur Elternklasse macht man in der Deklaration der Kindklasse dadurch deutlich, dass man hinter den Klassennamen einen Doppelpunkt (:), den Zugriffsspezifizierer `public` und den Namen der Elternklasse schreibt. Auf andere Zugriffsspezifizierer als `public` gehen wir im Rahmen des Praktikums nicht ein, da sie kaum verwendet werden.

Die Syntax zeigen die Codebeispiele in Listing 56 und Listing 57.

```
1   // Picture.h
2   // Deklaration der Elternklasse
3   class Picture
4   {
5   private:
6       int m_iHeight;
7       int m_iWidth;
8
9   public:
10      Picture();
11      // get und set Methoden, um die Attribute ein- bzw. auszulesen.
12      int getHeight();
13      void setHeight(int iHeight);
14      int getWidth();
15      void setWidth(int iWidth);
16
17      void transformToMonochrome();
18  };
```

Listing 55: Deklaration der Elternklasse


```
1  // BMP.h
2  #include “Picture.h”
3
4  // Deklaration der Klasse BMP als Kinderklasse der Klasse Picture
5  class BMP : public Picture
6  {
7  public:
8     void zoom();
9  };
```

Listing 56: Deklaration der Kindklasse BMP (Die Definition von BMP wird hier nicht dargestellt)

```
1  // JPG.h
2  #include “Picture.h”
3
4  // Deklaration der Klasse JPG als Kinderklasse der Klasse Picture
5  class JPG : public Picture
6  {
7  private:
8     float m_fCompressionFactor();
9  public:
10    void compressForEMail();
11 };
```

Listing 57: Deklaration der Kindklasse JPG

```
1  // JPG.cpp
2
3  // Definiton der Klasse JPG als Kindklasse der Klasse Picture
4  #include “JPG.h”
5
6  JPG::JPG(): Picture()                    // Aufruf von Konstruktor Picture über Initialisierungsliste
7  { /* ... */ }
8  float JPG::m_fCompressionFactor(){ /* ... */ }
9  void JPG::compressForEMail(){ /* ... */ }
```

Listing 58: Definition der Kindklasse JPG

Verständnisfragen

- Was versteht man unter Vererbung?
- Was ist eine Elternklasse?
- Was ist eine Kindklasse?
- Worin liegt der Nutzen der Vererbung?

1.17.3 Überschreiben

Wie im Abschnitt 1.17.1 bereits beschrieben, enthält die Kindklasse alle Attribute und Methoden, die ihre Elternklasse ihr vererbt hat. Außerdem kann sie selbst weitere definieren.

Allerdings zeichnet sich die Kindklasse nicht nur durch hinzukommende Merkmale aus. Es ist auch möglich, dass sie die von der Elternklasse vererbten Methoden modifiziert. Diesen Prozess nennt man Überschreiben.

Überschreiben geschieht, indem die Kindklasse eine vererbte Methode nochmals deklariert und neu definiert.

Hinweis: Überschreiben und Überladen

Überschreiben und Überladen dürfen auf keinen Fall verwechselt werden:

Überschreiben bezeichnet die Redefinition einer vererbten Methode in der Kindklasse. Die Signaturen der Methoden in Eltern und Kindklassen unterscheiden sich nicht, jedoch die Definitionen.

Überladen bezeichnet die Definition mehrerer Methoden mit demselben Namen jedoch unterschiedlichen Parametern. Die Methoden haben also verschiedene Signaturen.

Ruft man nur diese Methode durch ein Objekt der Kindklasse auf, so wird die überschriebene Methode verwendet. Ein Aufruf der Methode durch ein Objekt der Elternklasse ruft die ursprüngliche Methode auf.

Aufgabe: Vererbung und Überschreibung

a) Erstellen Sie die Klasse `Person` wie in Abbildung 16 beschrieben.

b) Erstellen Sie die Klasse `Student` und leiten Sie diese von `Person` ab, siehe Abbildung 16.

c) Überschreiben Sie die Methode `getName()` der Klasse `Student`, sodass „Student" vor dem Namen steht.

d) Erstellen Sie eine Main-Funktion, die jeweils ein Objekt der Klasse Student und eins der Klasse Person erstellt. Lassen Sie sich jeweils den Namen in die Konsole ausgeben, indem Sie die Funktion `getName()` verwenden.

e) Erstellen Sie sich nun ein Objekt der Elternklasse Person und überschreiben Sie dies mit dem in d) erstellen Objekt der Kindklasse Student. Überlegen Sie sich, wieso und wie dies funktioniert und welche Funktionen im Objekt von Person nun genutzt werden können. Überlegen Sie sich außerdem, ob Sie auch einem Objekt der Kindklasse ein Objekt der Elternklasse zuordnen können.

Verständnisfragen

- Worin liegt der Unterschied zwischen Überschreibung und Überladung?
- Worin liegt der Nutzen der Überschreibung?

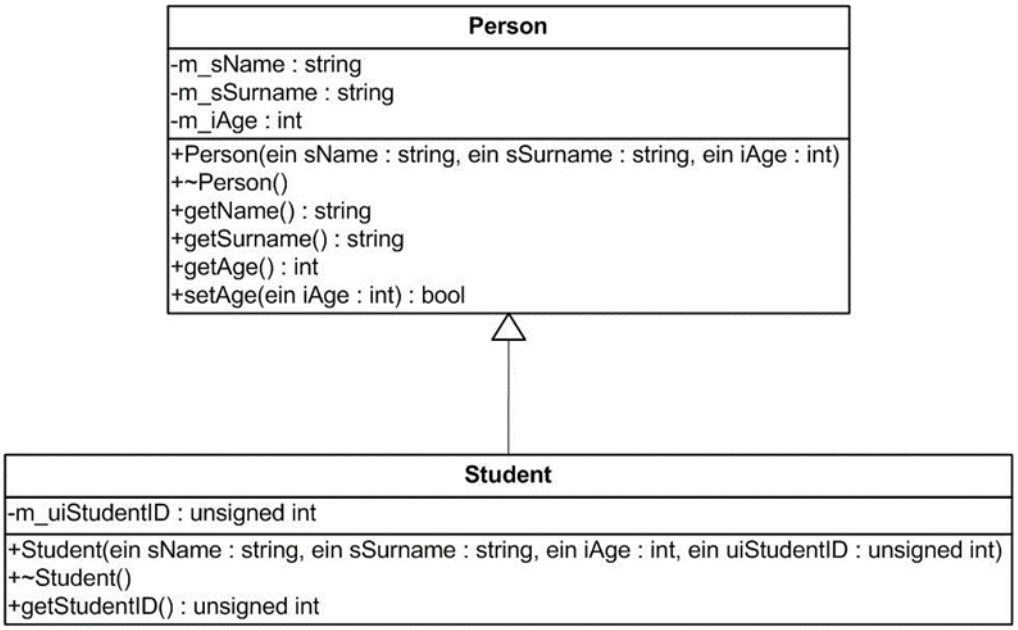


Abbildung 16: Ableitung der Klasse Student von der Klasse Person

1.17.4 Schnittstellen

Viele Programme lassen sich durch Module erweitern oder verändern. Denken Sie beispielsweise an Add-Ons für Internetbrowser, Windows Servicepacks oder Apps für Smartphones. Damit ein Programm beliebig viele Module nutzen kann und diese untereinander austauschbar sind, bedarf es eindeutiger Schnittstellen. Solche Schnittstellen lassen sich in der Objektorientierung sehr gut mithilfe von Interfaces (Schnittstellen) umsetzen. Eine Schnittstelle in C++ ist eine Klasse, von der sich keine Objekte bilden lassen, weil keine ihrer Methoden definiert ist. Die Methoden sind nur in der Headerdatei deklariert. Aus diesem Grund benötigt die Schnittstelle keinen Konstruktor und Destruktor. Die cpp-Datei entfällt.

Methoden, die nicht definiert sind, nennt man *rein virtuelle Methoden*. Der folgende Codeausschnitt zeigt, wie eine solche Methode deklariert wird.

```
1  // Rein virtuelle Methode deklarieren
2  virtual bool checkForUpdates() = 0;
```

Rein abstrakte Klassen sind Klassen, die nur rein virtuelle Methoden beinhalten und somit immer Elternklassen sind. Ihre Kinder müssen alle (rein virtuellen) Methoden überschreiben, damit man von ihnen Objekte bilden kann. Die rein abstrakte Klasse erzwingt also die Implementierung all ihrer Methoden in der Kindklasse. Die Signaturen sind durch rein abstrakte Klassen bereits vorgegeben. Die Kindklassen können natürlich noch zusätzliche Attribute und Methoden besitzen.

Der große Vorteil der Schnittstellen liegt darin, dass ein Programm über eine solche Schnittstelle mit vielen Modulen auf definierte Weise kommunizieren kann. Diese Kommunikation geschieht durch den Aufruf der Methoden, die in der Schnittstelle deklariert sind.

Die Deklaration der Schnittstellenklassen und deren Verwendung können Sie den Codebeispielen in Listing 59 bis Listing 63 entnehmen.

```
1  // AddOn.h
2  #ifndef ADDON_H
3  #define ADDON_H
4
5  #include "Settings.h"
6
7  // Deklaration der abstrakten Schnittstellenklasse
8  class AddOn
9  {
10 public:
11    // Deklaration der rein virtuellen Methoden
12    virtual bool checkForUpdates() = 0;
13    virtual bool install() = 0;
14    virtual bool uninstall() = 0;
15    virtual bool configure(Settings newSetting) = 0;
16    virtual void showHelp() = 0;
17    virtual void showMenue() = 0;
18 };
19 #endif
```

Listing 59: Deklaration der Schnittstelle

```
1  // PasswordManager.h
2  #ifndef PASSWORDMANAGER_H
3  #define PASSWORDMANAGER_H
4
5  #include <string>
6  #include "addon.h"
7
8  class PasswordManager:
9     public AddOn
10 {
11 private:
12    // Zusätzliche, nicht vererbte Attribute der Klasse PasswordManager
13    std::string m_sMasterPassword;
14
15    public:
16    PasswordManager();
17    ~PasswordManager();
18    bool checkForUpdates();
19    bool install();
20    bool uninstall();
21    bool configure(Settings newSetting);
22    void showHelp();
23    void showMenue();
24
25    // Zusätzliche, nicht vererbte Methoden der Klasse PasswordManager
26    void savePassword(string sPassword);
27 };
28 #endif
```

Listing 60: Deklaration der Kindklasse PasswordManager

```
1  // WeatherToolbar.h
2  #ifndef WEATHERTOOLBAR_H
3  #define WEATHERTOOLBAR_H
4
5  #include "Settings.h"
6  #include "AddOn.h"
7
8  class WeatherToolbar: public AddOn
9  {
10 public:
11    WeatherToolbar();
12    ~WeatherToolbar();
13    bool checkForUpdates();
14    bool install();
15    bool uninstall();
16    bool configure(Settings newSetting);
17    void showHelp();
18    void showMenue();
19
20 };
21
22 #endif
```

Listing 61: Deklaration der Kindklasse WeatherToolbar

```
1  #include "PasswordManager.h"
2
3  PasswordManager::PasswordManager()
4  { /* ... */ }
5
6  PasswordManager::~PasswordManager()
7  { /* ... */ }
8
9  bool PasswordManager::checkForUpdates()
10 {
11    // Implementierung
12    return true;
13 }
14
15 bool PasswordManager::install()
16 {
17    // Implementierung
18    return true;
19 }
20
21 // …
```

Listing 62: Definition einer Kindklasse

```
1   // main.cpp
2   #include "PasswordManager.h"
3   #include "AddOn.h"
4   #include "WeatherToolbar.h"
5
6   int main()
7   {
8       // Anlegen einer Liste von Pointern auf die AddOns
9       AddOn* AddOnList[2];
10
11      // Dynamische Erstellung der AddOns
12      // (siehe Hinweiskasten "Pointer vom Typ der Elternklasse auf eine Kindklasse")
13      AddOnList[0] = new PasswordManager;
14      AddOnList[1] = new WeatherToolbar;
15
16      // Die unterschiedlichen AddOn Klassen
17      // werden einfach über Pointer der Elternklasse angesprochen
18      for(int i = 0; i<2; i++)
19      {
20              AddOnList[i]->install();
21      }
22
23      return 0;
24  }
```

Listing 63: Verwendung der Kindklasse im Hauptprogramm

Hinweis: Pointer vom Typ der Elternklasse auf eine Kindklasse

Eine Besonderheit der Vererbung ist die Möglichkeit, Pointer vom Typ der Elternklasse auf Kindklassen zu erstellen.

```
ParentClass* pChild = new ChildClass()
```

Der Pointer der Elternklasse kann auf die vererbten Methoden und Attribute zugreifen.

Welchen Vorteil diese Eigenheit hat, zeigt das Codebeispiel in Listing 63. Statt jede von `AddOn` abgeleitete Klasse einzeln anzusprechen, kann man sie nun alle über die Elternklasse `AddOn` ansprechen. Dies führt zu einer Verringerung des Implementierungsaufwands und zu einer einfachen Integration neuer Klassen, die die Schnittstelle implementieren.

Wie Sie anhand des Beispiels gesehen haben, kann hinter der Elternklasse eine Vielzahl von Kindklassen stehen. Die von der Elternklasse vererbten Methoden können auf unterschiedlichster Weise implementiert werden. Diese Vielgestaltigkeit der vererbten Methoden in den unterschiedlichen Kindklassen nennt man Polymorphie.

1.17.5 Abstrakte Klassen

Die abstrakte Klasse enthält, genau wie die rein abstrakte Klasse, rein virtuelle Methoden. Der Unterschied zur rein abstrakten Klasse liegt darin, dass sie auch implementierte Methoden enthält. Eine abstrakte Klasse enthält mindestens eine rein

virtuelle Methode. Dadurch kann auch von ihr kein Objekt erzeugt werden. Folglich benötigt sie unter Umständen keinen explizit definierten Konstruktor und Destruktor.

1.17.6 Virtuelle Methoden

Bisher wurden nur rein virtuelle Methoden verwendet. Es gibt aber auch virtuelle Methoden. Virtuelle Methoden benötigt man nur dann, wenn man mit Pointern vom Typ der Elternklasse auf die Kindklasse arbeitet (siehe dazu den vorangegangen Hinweiskasten „Pointer vom Typ der Elternklasse auf eine Kindklasse“). Betrachten Sie das Codebeispiel in Listing 64 bis Listing 66.

```
1  // Parent.h
2  class Parent
3  {
4  public:
5      Parent();
6      ~Parent();
7      void method();
8      virtual void virtual_method();
9  };
```

Listing 64: Deklaration der Elternklasse

```
1  // Child.h
2  #include "parent.h"
3
4  class Child: public Parent
5  {
6  public:
7      Child();
8      ~Child();
9
10     // Überschriebene Methode
11     void method();
12     void virtual_method();
13 };
```

Listing 65: Deklaration der Kindklasse

```
1  // main.cpp
2  #include "Child.h"
3
4  void main ()
5  {
6      // Dynamisch Erstellung eines Child Objektes.
7      // Der Zugriff erfolgt über einen Pointer vom Typ der Elternklasse.
8      Parent* pChild = new Child();
9
10     // Es wird die Methode der Elternklasse nicht die überschriebene der Kindklasse aufgerufen?
11     pChild->method();
12     // Es wird die überschrieben Methode der Kindklasse aufgerufen.
13     pChild-> virtual_method();
14 }
```

Listing 66: main-Funktion

Wir haben eine in der Elternklasse definierte Methode in der Kindklasse überschrieben. Rufen wir nun diese Methode über einen Pointer vom Typ der Elternklasse, der auf die Kindklasse zeigt, auf, so ist zur Kompilierzeit noch nicht bekannt, ob die Methode der Eltern- oder der Kindklasse verwendet wird.

Da der Pointer vom Typ der Elternklasse ist, wird auch die Methode der Elternklasse verwendet, obwohl sie in der Kindklasse überschrieben wurde.

Um dennoch die Methode der Kindklasse aufzurufen, benötigt man das Schlüsselwort `virtual`. Stellt man dieses bei der Deklaration in der Elternklasse der Methode voran, so wird die Elternklassenmethode nur dann verwendet, wenn sie in der Kindklasse nicht überschrieben ist.

Aufgabe: Virtuelle Methoden

Verwenden Sie für diese Aufgabe die bereits implementierten Klassen *Person* und *Student*.

a) Deklarieren Sie mehrere Pointer auf Person.

b) Weisen Sie diesen Pointern dynamisch erzeugte Objekte der Klasse Student zu. Geben Sie in einer main-Funktion die Eigenschaften der Objekte in der Konsole aus.

c) Implementieren Sie nun die Methoden der Klassen so, dass die Studentenobjekte Ihre überschriebene Methode benutzen. Geben Sie die Objekte erneut in der Konsole aus.

Aufgabe: Polymorphie

Lesen Sie den Quellcode in Listing 67 durch. Welche Klassen und Funktionen gibt es und wie hängen sie zusammen?

Die folgenden Aufgaben sind durch Überlegen und das handschriftliche Notieren der Ausgabe zu lösen. Beachten Sie den Hinweis.

a) Schauen Sie sich nun die main-Funktion an (Listing 68): Was geschieht in den Zeilen 30-32? Werden sie fehlerfrei kompiliert? Wenn ja, zu welcher Ausgabe in der Konsole führen die einzelnen Anweisungen?

b) Was geschieht in den Zeilen 34-37? Welche Anweisungen können nicht kompiliert werden und warum? Wie sieht die Konsolenausgabe dieser Zeilen aus? Ist es besser, Objekte oder Pointer zu verwenden, wenn man mit polymorphen Klassen arbeitet und keine Informationen verlieren möchte?

Achtung: Bitte Listing 67 und Listing 68 beachten. Aufgabe ist auf Papier zu lösen.

Hinweis: Polymorphie

Polymorphie in C++ bedeutet, dass ein Funktionsaufruf unterschiedliche Funktionen ausführen kann, je nachdem welche Objekttyp die Funktion aufruft.

Soll ein Objekt der Kindklasse eine überschriebene Methode aufrufen, so kann diese wie in der Aufgabe „Vererbung und Überschreibung“ definiert und aufgerufen werden (siehe Zeile 30 in Listing 68).Problematisch wird es, wenn das Objekt nicht direkt für den Aufruf zur Verfügung steht, sondern nur mittels Funktionsname aufgerufen wird (siehe Zeile 8 in Listing 67). In diesem Fall wird die Methode der Elternklasse aufgerufen. **Außer** diese wird als **virtual** deklariert. Dieses Schlüsselwort ermöglicht die Nutzung der überschriebenen Methode aus der Kindklasse.

```
1   // Polymorphie.h
2   // alle Methoden wurden in der Headerdatei implementiert, da sie sehr kurz sind,
3   // es existiert also keine .cpp Datei
4   #include <iostream>
5
6   class A{
7   public:
8       virtual void g(){std::cout << "A.g\n"; f();}
9       virtual void f(){std::cout << "A.f\n"; h();}
10      void h(){std::cout << "A.h\n"; i();}
11      virtual void i(){std::cout << "A.i\n";}
12  };
13
14  class B: public A{
15  public:
16      void g(){std::cout << "B.g\n"; f();}
17      void f(){std::cout << "B.f\n"; A::f(); h();}
18  };
19
20  class C: public B{
21  public:
22      void g(){std::cout << "C.g\n"; f();}
23      void h(){std::cout << "C.h\n"; i();}
24      void i(){std::cout << "C.i\n";}
25  };
```

Listing 67: Vererbung und Polymorphie: Headerdatei mit Implementierung

```
26  // main.cpp
27  #include “Polymorphie.h”
28  int main()
29  {
30      A a = A(); a.f();
31      B b = B(); b.g();
32      C c = C(); c.g();
33
34      B b0 = A(); b0.g();
35      A a1 = B(); a1.f();
36      A a2 = b; a2.f();
37      A* a3 = &b; a3->f();
38      return 0;
39  }
```

Listing 68: Vererbung und Polymorphie: main.cpp

Verständnisfragen

- Worin liegt der Unterschied zwischen *Schnittstellen* und *abstrakten* Klassen?
- Was versteht man unter einer *rein virtuellen* Methode?
- Was versteht man unter einer *virtuellen* Methode?
- Was versteht man unter Polymorphie?

1.18 Rekursion

1.18.1 Einleitung

Algorithmen lassen sich auf zwei Arten beschreiben, iterativ und rekursiv. Den iterativen Ansatz haben Sie bereits kennengelernt; in diesem Kapitel soll nun die rekursive Programmierung verwendet werden.

Während bei Iteration die Programmteile nacheinander abgearbeitet werden, ruft sich bei einer Rekursion die Funktion immer wieder selbst auf, bis ein bestimmtes Abbruchkriterium erreicht wird. Da die aufrufende Funktion warten muss, bis die aufgerufene Funktion das Ergebnis zurückliefert, wächst der call stack stetig an. Erst wenn die aufgerufenen Funktionen ihren Wert zurückliefern, werden die Funktionen und ihre Daten vom Stack entfernt.

1.18.2 Anwendung

Um Ihnen die Rekursion näher zu bringen, werden Sie die Fakultät einer gegebenen Zahl berechnen.

Die Fakultät wird durch

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1! \cdot 0!$$

mit

$$0! = 1! = 1$$

berechnet. Für negative Werte ist die Fakultät nicht definiert und die Funktion soll in diesem Fall 0 zurückliefern.

Die Berechnung der Fakultät ist – im Gegensatz zu anderen Problemstellungen – sowohl iterativ als auch rekursiv möglich. Beim iterativen Ansatz multiplizieren Sie unter Verwendung einer Schleife. Der rekursive Ansatz hingegen unterteilt sich in mehrere Schritte, die an der Berechnung von 5! gezeigt werden.

- Zur Berechnung von 5! wird das Ergebnis von 4! mit 5 multipliziert.

$$5! = 5 \cdot 4!$$

- Das Ergebnis für 4! liegt aber noch nicht vor. Daher ruft sich die Funktion erneut auf. Diesmal wird als Argument 4 übergeben.
- Zur Berechnung von 4! wird das Ergebnis von 3! mit 4 multipliziert.

$$4! = 4 \cdot 3!$$

- Da auch hier das Ergebnis noch nicht vorliegt, ruft sich die Funktion abermals erneut auf, jedoch mit 3 als Parameter.
- Diese Schritte werden solange durchgeführt, bis eine Abbruchbedingung erfüllt ist. Hier ist sie für n=1 erfüllt, d.h. die Funktion beendet sich, indem 1 über `return` zurückgegeben wird.
- Das Ergebnis wird für die aufrufenden Funktionen verwendet, usw., bis schließlich die erste aufgerufene Funktion (für 5!) ihr Ergebnis zurückliefert.

Aufgabe: Rekursion

a) Implementieren Sie zuerst die Berechnung der Fakultät iterativ. Erstellen Sie hierfür eine Funktion `factorialIter`.

b) Implementieren Sie nun die Berechnung der Fakultät rekursiv. Erstellen Sie hierfür eine Funktion `factorialRec`.

Aufgabe: Fibonacci-Folge

Die Fibonacci-Folge ist eine unendliche Folge, in der das jeweils nächste Folgenglied rekursiv aus den zwei vorhergehenden Gliedern berechnet wird:

$$f_n = f_{n-1} + f_{n-2} \quad \text{für} \quad n > 2$$

Wobei $f_1 = f_2 = 1$

a) Schreiben Sie eine Funktion, welche für ein übergebenes n das n-te Glied der Fibonacci-Folge berechnet.

b) Geben Sie die ersten 15 Glieder der Fibonacci-Folge in der Konsole aus (z.B. mittels for-Schleife).

Verständnisfragen

- Was versteht man unter Rekursion?
- Worin liegt der Unterschied zu Iteration?

- Welcher Algorithmus lässt sich nicht iterativ beschreiben? Nennen Sie ein Beispiel.

Zusatzaufgabe: Springerproblem

Beim sogenannten Springerproblem handelt es sich um ein kombinatorisches Problem, bei welchem ein Springer auf einem (8x8) Schachbrett eine Route finden soll, bei der er jedes Feld genau einmal besucht. Im Folgenden soll ein Backtracking-Algorithmus implementiert werden, der das Problem Schritt für Schritt löst.

a) Überlegen Sie sich eine angemessene Darstellung für das Schachbrett. Dieses sollte leicht an andere Funktionen übergeben werden können und es sollte ersichtlich sein, ob ein Feld schon besucht wurde oder nicht.

b) Schreiben Sie eine Funktion, die das Schachbrett und ein Feld übergeben bekommt und zurückgibt, ob das übergebene Feld wirklich auf dem Schachbrett liegt.

c) Schreiben Sie eine rekursive Funktion `findpath(Schachbrett; Feldindex1; Feldindex2)`, die mit Hilfe von Backtracking das Schachbrett nach einem gültigen Pfad durchsucht.

d) Testen Sie das Programm und geben Sie einen gültigen Pfad in der Konsole aus.

Achtung: Besser erst am Ende des Tages lösen!

1.19 Dateien lesen und schreiben

C++ stellt die folgenden Klassen (Streams) zur Verfügung, um Zeichen in Dateien zu schreiben oder aus Dateien auszulesen:

- *ofstream*: Klasse/Stream, um Dateien zu schreiben
- *ifstream*: Klasse/Stream, um Dateien zu lesen
- *fstream*: Klasse/Stream, um Dateien zu schreiben und zu lesen

Um die Klassen verwenden zu können, muss die Bibliothek `fstream` mit dem Befehl `#include <fstream>` eingebunden werden. Die drei Klassen gehören zum Standard-Namespace und sind direkt oder indirekt von den Klassen `istream` und `ostream` abgeleitet, welche wir (unbemerkt) schon kennengelernt haben: `std::cout` ist vom Typ `ostream` und `std::cin` vom Typ `istream`. Dementsprechend funktioniert auch die Verwendung der Filestreamklassen `ofstream`, `ifstream` und `fstream` analog zu der von `std::cout` und `std::cin`. Einziger Unterschied ist, dass Filestreams mit den Dateien verknüpft werden, die sie bearbeiten. Listing 69 zeigt die Verwendung eines `fstreams`.

```
1   // Einfache Dateioperationen
2   #include<iostream>
3   #include<fstream>
4
5
6   int main()
7   {
8       std::fstream file;                  // Deklaration eines Objekts vom Typ fstream
9       file.open("beispiel.txt");          // file wird mit einer Datei assoziiert
10      file << "Beispieltext in Datei";    // file schreibt einen Satz in die Datei
11      file.close();
12
13      return 0;
14  }
```

Listing 69: Schreiben von Dateien

Obiger Code erstellt eine Datei mit dem Namen beispiel.txt, öffnet sie, schreibt einen Text hinein und schließt sie wieder. Jede dieser Operationen wird im Folgenden ausführlich erklärt:

1.19.1 Öffnen einer Datei

Um in eine Datei schreiben oder sie auszulesen, muss sie zunächst geöffnet werden. Das geschieht über die Member Funktion:

`open(filename, mode).`

filename ist ein C-string mit dem Namen der zu öffnenden Datei. Wenn ein string aus C++ als Dateiname verwendet werden soll, so muss dieser mit der Methode `c_str()` in einen C-string umgewandelt werden. Listing 70 zeigt die Vorgehensweise.

```
1  // Umwandlung eines C++-strings in einen C-string
2
3  std::string filename;                  // C++ string
4  // file.open(filename.c_str());        // Umwandlung in C-String (nötig vor C++ 11)
5  file.open(filename);                   // Funtkionsaufruf seit C++ 11
```

Listing 70: C++-string in C-string umwandeln

mode ist ein optionaler Parameter, der aus einer Kombination von verschiedenen „Flags“ besteht. Die Flags werden über den bitweisen OR-Operator „|“ kombiniert. Tabelle 6 zeigt die möglichen Flags.

Modus	Abgek.	Bedeutung
`ios::in`	**in**put	Öffnen einer Datei mit lesendem Zugriff
`ios::out`	**out**put	Öffnen einer Datei mit schreibendem Zugriff
`ios::ate`	**at e**nd	Setzt den Pointer zum Lesen/Schreiben ans Ende der Datei. Kann aber auch an andere Positionen verlegt werden um dort zu Lesen oder Schreiben
`ios::app`	**app**end	Neuer Text wird am Ende der bestehenden Datei angehängt
`ios::trunc`	**trunc**ate	Vorhandener Inhalt einer Datei wird gelöscht und neuer Inhalt wird hineingeschrieben

Tabelle 6: Verschiedene Modi für den Dateizugriff

Will man zum Beispiel eine Datei mit Lese- und Schreibzugriff öffnen und den bisherigen Inhalt überschreiben, erzielt man dies durch:

```
1  // Datei öffnen mit Lese-und Schreibzugriff. Bisheriger Inhalt wird überschrieben.
2  std::fstream file;
3  file.open("beispiel.txt", std::ios::in | std::ios::out | std::ios::trunc);
```

Listing 71: Beispiel eines Dateizugriffs. Lese-und Schreibzugriff, Inhalt überschreiben

Mit dem Befehl `filestream.is_open()` kann man überprüfen, ob eine Datei erfolgreich von einem Filestream geöffnet wurde:

```
1  // Feststellen, ob eine Datei geöffnet ist
2  if(file.is_open()){ /* Datei ist geöffnet! -> Fortfahren… */ }
```

Listing 72: Feststellen ob eine Datei geöffnet ist

1.19.2 Schließen einer Datei

Ist die Bearbeitung einer Datei abgeschlossen, so muss der Befehl `filestream.close()` aufgerufen werden. Es werden die Ressourcen und der Filestream freigegeben und die Dateien können erneut verwendet werden.

1.19.3 Lesen und Schreiben einer Datei

Das Lesen und Schreiben von Dateien erfolgt analog zur Verwendung von `std::cin` und `std::cout`. Die Shiftoperatoren `<<` und `>>` können auch auf Filestreams angewendet werden. Listing 73 zeigt die Verwendung.

```
1   // Lesen und Schreiben von Dateien
2   #include <iostream>
3   #include <fstream>
4   #include <string>
5
6   int main()
7   {
8      std::string str = “Teststring“;
9      int iNumber = 10;
10
11     std::ofstream ofile;                                  // Filestream mit Schreibzugriff
12     ofile.open("beispiel.txt", std::ios::out | std::ios::trunc);  // alter Dateiinhalt wird gelöscht
13     ofile << iNumber <<  std::endl  << str << std::endl;  // schreiben von num und str in Datei
14     ofile.close();
15
16     iNumber = 0; str = "";                                // zurücksetzen von num und str
17
18     std::ifstream ifile;                                  // Filestream mit Lesezugriff
19     ifile.open("beispiel.txt");
20     ifile >> iNumber;                                     // wiedereinlesen von num
21     ifile >> str;                                         // wiedereinlesen von str bis zum
22                                                          // ersten Leerzeichen oder
23                                                          // Zeilenumbruch
24     ifile.close();
25
26     return 0;
27  }
```

Listing 73: Lesen und Schreiben von Dateien

1.19.4 Zustände einer Datei

Von `std::cin` wissen wir schon, dass es verschiedene Eigenschaften hat, die den momentanen Zustand des Streams beschreiben. Dasselbe gilt für Filestreams. Tabelle 7 zeigt die Zustände, die ein Filestream annehmen kann:

Zustand	Bedeutung
`good()`	Gibt an, ob der Stream fehlerfrei und nicht am Dateiende ist
`eof()`	Gibt an, ob das Ende der Datei erreicht wurde
`fail()`	Gibt an, ob das Failbit gesetzt wurde. Das Failbit wird gesetzt, wenn ein logischer oder ein Lese/Schreibfehler aufgetreten ist
`clear()`	Setzt das Failbit auf false zurück

Tabelle 7: Mögliche Zustände eines Filestreams

Hinweis: Die Methode `getline`

In der Bibliothek `<string>` gibt es die Methode

```
&istream getline(istream& is, string& str)
```

die es ermöglicht, eine gesamte Zeile aus einem Filestream in eine Stringvariable zu kopieren. Die Methode wird verwendet wie in Listing 74 gezeigt.

```
1   #include <string>
2   #include <iostream>
3
4   int main()
5   {
6       std::string sName;
7       std::cout << "Bitte vollständigen Namen eingeben";
8       getline(std::cin, sName);
9       // Speichern einer ganzen Zeile der Konsoleneingabe in name
10      // Bei Auslesen aus einer Datei den geöffneten FileStream anstatt cin verwenden)
11      std::cout << "Ihr Name ist: " << sName << std::endl;
12
13      return 0;
14  }
```

Listing 74: Verwendung der Funktion `getline`

Aufgabe: Dateien lesen und schreiben

Schreiben Sie ein Programm, das eine Textdatei auslesen bzw. schreiben kann. Gehen Sie dazu folgendermaßen vor:

a) Schreiben Sie eine Funktion `read_file(string Dateiname)`, die die übergebene Datei öffnet und zeilenweise ausliest. Jede Zeile soll in die Konsole ausgegeben werden. Wurde die Datei bis ans Ende gelesen, wird sie wieder geschlossen.

b) Schreiben Sie eine Funktion `write_file(string Dateiname)`, die die übergebene Datei so zum Schreiben öffnet, dass der Text ans Ende der Datei angehängt wird. Daraufhin soll der Benutzer zeilenweise Eingaben in die Konsole machen können, welche die Funktion an die Datei anhängt. Bei dem Befehlt `exit` soll das Schreiben der Datei beendet und die Funktion verlassen werden.

c) Legen Sie im Projektordner eine Textdatei (.txt) an und füllen Sie sie mit beliebigem Text. Schreiben Sie eine main-Funktion, die vom Benutzer einen Dateinamen einliest. Der Inhalt der Datei wird in die Konsole ausgegeben. Anschließend erhält der Benutzer die Möglichkeit, neuen Text an die Datei anzuhängen. Abschließend wird die Datei erneut in die Konsole ausgegeben.

1.20 Fehlerbehandlung

Im Kapitel über Konsolen Ein- und Ausgabe wurde gezeigt, wie man in einem Programm auf (spezielle) Fehler reagieren kann, z.B. auf eine falsche Benutzereingabe. In diesem Kapitel wird nun ein universelles Konzept zur Fehlerbehandlung in C++ vorgestellt; das Konzept der *Ausnahme* (*Exception*).

1.20.1 Ausnahmen

Die Fehlerbehandlung durch Ausnahmen hat folgendes Prinzip: Wenn eine Programmkomponente (z.B. eine Methode), auf ein auftretendes Problem nicht angemessen reagieren kann, kann sie eine Ausnahmesituation erzeugen (`throw object`). Diese Ausnahmesituation kann dann eventuell eine andere Komponente behandeln. Welche Fehler eine bestimmte Komponente abfangen kann zeigt sie durch die „Catch“ Abfrage.

Im Detail sieht das wie folgt aus:

- Möchte eine Komponente einen bestimmten Fehler behandeln, so wird der kritische Codeabschnitt, in dem Fehler auftreten können, in einem `try{}`-block eingeschlossen. Die entsprechenden Fehler können in einem anschließenden `catch{}`-block abgefangen und behandelt werden.
- Tritt in einer Komponente ein Fehler auf, den sie nicht selbst behandeln kann, so bricht sie ihre Ausführung ab und wirft eine Ausnahme (`throw object`). Diese Ausnahme wird an die aufrufende Komponente weitergeben. Dabei enthält `object` Informationen über den aufgetretenen Fehler.

Listing 75 zeigt die Verwendung von Ausnahmen:

```
1   // Fehlerbehandlung durch Ausnahmen
2   void taskmaster()
3   {
4       try
5       {       // faengt auftretende Fehler ab
6               int iResult = doTask();
7               // mit result weiterarbeiten
8       }
9       catch(SomeError){
10              // Fehler aufgetreten: Hier erfolgt die Fehlerbehandlung
11      }
12  }
13
14  int doTask()
15  {
16      if( /* Abfrage */ )
17              return iResult;
18      else
19              throw SomeError{};
20  }
```

Listing 75: Fehlerbehandlung mit Ausnahmen

Die Methode `taskmaster()` ruft `doTask()` auf, um eine Berechnung auszuführen. Wenn `doTask()` die Berechnung erfolgreich ausführt, gibt sie das Ergebnis zurück. Ist die Berechnung nicht erfolgreich, so wirft `doTask()` eine Ausnahme, im Beispiel das Objekt: `SomeError{}`. Das Werfen der Ausnahme zeigt `taskmaster()`, dass die Berechnung nicht funktioniert hat. Im `catch(SomeError)`-Block kann dann auf den Fehlertyp `SomeError` reagiert werden.

1.20.2 Der `catch`-Block

In den meisten Programmen können verschiedene Fehlertypen auftreten. In diesem Fall können mehrere `catch`-Blöcke für verschiedene Fehler an einen `try`-Block angefügt werden. Tritt ein Fehler auf so wird das Programm im ersten passenden `catch`-Block fortgeführt.

Daraus ergibt sich, dass man bei der Anordnung von `catch`-Blöcken achtgeben muss. Es müssen zunächst spezielle Fehler und erst dann allgemeine Fehler abgefangen werden, da sonst ggf. ein `catch`-Block nie erreicht wird.

Um alle in einem `try`-Block auftretenden Fehler abzufangen, gibt es den Block:

`catch(...){/*Anweisungen*/}.`

Listing 76 und Listing 77 geben ein falsches und ein korrektes Beispiel an:

```
1  // Falsche Anordnung von catch-Blöcken
2  try
3  {
4     // Kritischer Codeblock
5  }
6  catch(…)
7  {
8     // Faengt ALLE Fehler ab
9  }
10 catch(Divide_by_zero)
11 {
12    // Problem: Dieser Block wird nie erreicht und deswegen nie ausgeführt!
13 }
```

Listing 76: Falsche Anordnung von catch-Blöcken

```
1  //Richtige Anordnung von catch-Blöcken
2  try
3  {
4     // kritischer Codeblock
5  }
6  catch(Divide_by_zero)
7  {
8     // Erst „spezielle“ Fehler abfangen
9     // Reagiere hier auf divide-by-zero-Fehler
10 }
11 catch(…) {
12    // Faengt alle anderen Fehler ab
13 }
```

Listing 77: Richtige Anordnung von catch-Blöcken

1.20.3 Ausnahmen und Ressourcen

Ausnahmen sind ein sehr effizientes Mittel um eine fehlerfreie Ressourcennutzung zu gewährleisten. Ein Beispiel:

```
1  // Moeglicher Ressourcenverlust
2  #include <fstream>
3
4  void doSth()
5  {
6     // Filestream mit Schreibzugriff
7     std::ofstream ofile;
8
9     // Datei öffnen
10    ofile.open("beispiel.txt");
11
12    // Datei bearbeiten…
13
14    // Datei schließen
15    ofile.close();
16 }
```

Listing 78: gefährliche Verwendung von Ressourcen

Tritt in Listing 78 irgendwo zwischen Zeile 6 und 10 ein Fehler auf, nachdem die Datei geöffnet wurde, so bricht die Ausführung von `doSth()` ab, bevor die Datei wieder geschlossen wird. Damit bleibt sie aber vom Programm geöffnet und andere Programme können nicht darauf zugreifen. Dies gilt auch für andere Ressourcen, z.B. wenn Speicherplatz über `new` zugeteilt wurde.

Um derartige Fehler zu vermeiden, können Ausnahmen verwendet werden. Listing 79 zeigt einen verlustsicheren Zugriff auf Ressourcen.

```
1  // Sicherer Dateizugriff
2  #include <fstream>
3
4  void doSth()
5  {
6     std::ofstream ofile;
7
8     // Fange Fehler beim Dateizugriff ab
9     try
10    {
11          ofile.open("beispiel.txt");
12          // Datei bearbeiten…
13          ofile.close();
14    }
15    catch(…)
16    {
17          // Schließt die Datei auf jeden Fall
18          if(ofile.is_open())
19          {
20              ofile.close();
21          }
22    }
23 }
```

Listing 79: sicherer Zugriff auf Ressourcen

1.20.4 Ausnahmeklassen aus der Standardbibliothek

Die C++ Standardbibliothek stellt die Basisklasse `exception` zur Verfügung, von der eigene Ausnahmeklassen abgeleitet werden können. So können sehr spezifische Ausnahmeklassen erstellt werden, die umfangreiche Informationen über einen

bestimmten Fehler enthalten. Um die Basisklasse `exception` zu verwenden, muss folgende Bibliothek eingebunden werden: `#include <exception>`

Listing 80 zeigt, wie man Ausnahmeklassen von `exception` ableitet und verwendet. In den Zeilen 6-12 wird eine neue Ausnahmenklasse erstellt. Diese ist von `exception` abgeleitet und überschreibt die virtuelle, konstante Methode `what()`. Der Rückgabe-Typ von `what()` ist `const char*` und das nachgestellte `throw()` ist ein Schlüsselwort, das dem Compiler anzeigt, dass diese Methode selbst keine Ausnahme wirft. In Zeile 12 wird ein Objekt der Klasse `myFatalError` mit dem Namen `myErr` erstellt.

In der main-Methode wird das Objekt `myErr` in einem `try`-Block geworfen und vom anschließenden `catch`-Block abgefangen. Dieser gibt dann die Rückgabe der Methode `what` in der Konsole aus.

```
1  // Benutzung von eigenen Ausnahmeklassen
2  #include <iostream>
3  #include <exception>
4
5  // Definition einer eigenen Ausnahmeklasse
6  class myFatalError : public std::exception
7  {
8     public:
9     virtual const char* what() const throw()
10    {
11          return "Fataler Fehler aufgetreten";
12    }
13 };
14
15 int main()
16 {
17    try
18    {
19          // … hier steht der kritische Codeteil und die Fehlerabfrage
20             throw myFatalError{};  // Wird „geworfen“ wenn fehler im kritischen Codeteil auftritt
21          // … Folgeanweisungen
22    }
23    catch (myFatalError& e)
24    {
25          std::cout << e.what() << std::endl;
26    }
27
28    return 0;
29 }
```

Listing 80: Definition und Verwendung eigener Ausnahmeklassen

Des Weiteren gibt es in der Standardbibliothek noch einige vordefinierte Ausnahmeklassen, die von `exception` abgeleitet wurden. Sie werden von einigen Komponenten der Standardbibliothek verwendet, wenn Fehler auftreten.

Ausnahme	Bedeutung
`bad_alloc`	wird von `new` geworfen, wenn ein Fehler bei der Speicherzuweisung auftritt

`bad_cast`	wird von `dynamic_cast` geworfen, wenn die Typumwandlung fehlschlägt
`bad_typeid`	wird von `typeid` geworfen, wenn ein Fehler auftritt

Tabelle 8: Einige Ausnahmen der C++ Standardbibliothek

Aufgabe: Ausnahmen

Erstellen Sie ein neues Projekt und kopieren Sie die `read_file` Methode der vorherigen Aufgabe in das neue Projekt. Diese Methode soll nun um Folgende Funktionalitäten erweitert werden:

a) Fügen Sie eine Klasse `nofileException` hinzu, die sich von `exception` ableitet und die Methode `virtual const char* what() const throw` überschreibt. Die Methode soll so überschrieben werden, dass sie die Fehlermeldung: „Die Datei existiert nicht“ zurückgibt. Erstellen Sie auch direkt ein Objekt der Klasse. Listing 80 zeigt das Vorgehen.

b) Kann eine Datei nicht geöffnet werden, so wird beim Aufruf von `fstream::open(…)` das Failbit (`fstream::fail()`) gesetzt. Verändern Sie die Methode `read_file(…)` so, dass sie das erstellte Objekt vom Typ `nofileException` wirft, wenn die Datei nicht geöffnet werden kann.

c) Rufen Sie in der main-Funktion die Methode `read_file` auf. Fügen Sie anschließend einen `try-catch`-Block ein, der Ausnahmen vom Typ `exception` abfängt (wie in Listing 80) und die Methode `what()` in der Konsole ausgibt, wenn ein Fehler in `read_file` auftritt.

1.21 Doppelt verkettete Listen

In der Standardbibliothek von C++ werden verschiedene Container zum Aufnehmen von Elementen eines oder mehrerer Typen bereitgestellt. In diesem Kapitel wird der generische Typ `list<typename T>` vorgestellt, der eine doppelt verkettete Liste repräsentiert.

Um `list<typename T>` verwenden zu können, muss die Bibliothek `<list>` aus der C++ Standardbibliothek eingebunden werden.

1.21.1 Allgemeines zu doppelt verketteten Listen

“Doppelt verkettetet“ bedeutet, dass jedes Element der Liste einen Pointer auf das vorhergehende und das nachfolgende Element enthält. So kann eine doppelt verkettete Liste sehr einfach in beide Richtungen (von vorne nach hinten sowie von hinten nach vorne) durchlaufen werden. Die erhöhte Flexibilität geht auf Kosten von Speicherplatz und Komplexität, die aber für einfache Anwendungen weniger relevant sind.

Die <>-Klammern nach list deuten an, dass es sich um einen generischen Typ handelt. Generische Typen sind dadurch charakterisiert, dass sie für verschiedene Typen implementiert sind. `Typename T` ist ein Platzhalter für den Typ, der verwendet werden soll.

Listing 81 zeigt wie Listen, die verschiedene Typen beinhalten implementiert werden.

```
1   #include <list>
2
3   class A{};
4
5   int main()
6   {
7       // Liste von int-Werten
8       std::list<int> myintlist;
9
10      // Liste von float Werten
11      std::list<float> myfloatlist;
12
13      // Liste von Objekten des Typs A
14      std::list<A> myAlist;
15
16      return 0;
17  }
```

Listing 81: Listen, die verschiedene Datentypen aufnehmen

1.21.2 Iteratoren

Mit einem Iterator wird eine Stelle in der Liste bezeichnet. Das Element an dieser Stelle kann dann zum Beispiel ausgelesen, verändert oder gelöscht werden. Außerdem können dort neue Elemente eingefügt werden. Ein Iterator wird über das Schlüsselwort `container::iterator` definiert. Die folgende Zeile definiert einen Iterator für eine `float`-Liste mit dem Namen `myiterator`:

```
std::list<float>::iterator myiterator;
```

Damit der Iterator benutzt werden kann, muss er initialisiert werden. Typischerweise initialisiert man einen Iterator auf die Position des ersten Elementes einer Liste. Die Funktion `listenname.begin()` liefert einen Iterator, der auf die Position des ersten Elements der Liste zeigt. Die Position hinter dem letzten Element der Liste liefert die Funktion `listenname.end()`.

Wird eine Liste mit Hilfe eines Iterators durchlaufen, so sind nach dem Durchlaufen der Laufparameter und der Rückgabewert von `listenname.end()` gleich. Die Liste wird durch Inkrementieren des Iterators durchlaufen. Auf ein Element der Liste wird über den `*`-Operator zugegriffen. Listing 82 zeigt das typische Vorgehen, um eine Liste (`List_1`) in einer `for`-Schleife zu durchlaufen.

```
1   std::list<float>::iterator myiterator
2
3   for(myiterator = list1.begin(); myiterator != list1.end(); myiterator++)
4   {
5       *(myiterator) += 2.5;
6   }
```

Listing 82: Durchlaufen einer Liste mit einem Iterator

1.21.3 Konstruktoren

Der Container `list` bietet mehrere Konstruktoren zur Initialisierung an. Die wichtigsten sind in Tabelle 9 aufgelistet.

Konstruktor	Bedeutung
`std::list<T> mylist;`	Erstellung einer leeren Liste
`std::list<float> mylist(n,val);`	Initialisiert `mylist` n Kopien von `val`
`std::list<float> mylist(list1.begin(), list1.end());`	Initialisiert `mylist` mit den Elementen [`list1.begin()`, `list1.end()-1`]
`std::list<float> mylist (list1);`	Kopierkonstruktor. Kopiert alle Elemente nach `mylist`

Tabelle 9: Konstruktoren von `list`

1.21.4 Methoden von `list`

Der Container `list` bietet umfangreiche Möglichkeiten um Elemente einzufügen, zu entfernen, anzuhängen, etc. Tabelle 10 listet die wichtigsten auf.

Operation	Bedeutung
`x = mylist.size()`	`x` erhält die Länge der Liste
`mylist.push_front(value)`	Erstellt Element mit Wert `value` am Anfang der Liste
`mylist.push_back(value)`	Erstellt Element mit Wert `value` am Ende der Liste
`mylist.insert(Iterator, value)`	Fügt Element mit Wert `value` vor der Position ein, auf die `Iterator` zeigt
`mylist.pop_front()`	Löscht das erste Element der Liste
`Iterator = mylist.erase(Iterator);`	Entfernt das Element aus der Liste, auf das `Iterator` zeigt. `erase(…)` gibt die Position des nächsten Elements an `Iterator` zurück
`mylist.erase(Iterator_1, Iterator_2);`	Löscht Elemente von `Iterator_1` (einschließlich) bis zu `Iterator_2` (ausschließlich). Nur `Iterator_2` bleibt gültig
`mylist.clear();`	Löscht die gesamte Liste

Tabelle 10: wichtigste Methoden von list

Listing 83 zeigt die Verwendung von `list`.

```
1   #include <list>
2
3   int main()
4   {
5       // Erstellen von Listen
6       std::list<float> flstList(3,2.0);                   // Liste mit 3 Elementen: 2.0 2.0 2.0
7       std::list<float> mylist(++flstList.begin(), flstList.end());   // Kopiert Elemente 2 und 3
8                                                           // mylist = 2.0 2.0
9       // Einfügen von Elementen
10      mylist.push_front(4.0);                             // mylist = 4.0 2.0 2.0
11      mylist.push_back(6.0);                              // mylist = 4.0 2.0 2.0 6.0
12      mylist.insert(++mylist.begin(),3.0);                // mylist = 4.0 3.0 2.0 2.0 6.0
13
14      // Löschen von Elementen
15      mylist.pop_front();                                 // mylist = 3.0 2.0 2.0 6.0
16
17      // Löschen mit Iterator
18      std::list<float>:: iterator myiter = mylist.begin();    // Iterator zeigt auf Anfang
19      myiter += 2;                                        // Iterator zeigt auf Element 3
20      myiter =mylist.erase(myiter);                       // Löscht Element 3
21
22      // Zugriff auf Elemente
23      std::list<float>::iterator myiterator;
24
25      for(myiterator = mylist.begin(); myiterator != mylist.end(); myiterator++)
26              *(myiterator) += 2.5;
27
28      // Löschen der gesamten Listen
29      flstList.clear();                                   // löscht Liste flstList
30      mylist.clear();                                     // löscht liste mylist
31
32      return 0;
33  }
```

Listing 83: Verwendung von `list`

Aufgabe: Liste von Koordinaten

Doppelt verkettete Listen sind sehr nützlich um eine Menge von Objekten aufzunehmen. Um dies zu demonstrieren, soll eine Liste erstellt und bearbeitet werden, die Koordinaten ((x,y)-Paare) aufnehmen kann.

a) Erstellen Sie eine `struct coord`, die eine Koordinate (ein (x,y)-Paar) repräsentiert. `x` und `y` sind vom Typ `float`. Das `struct` soll einen Konstruktor haben, der `x` und `y` mit übergebenen Werten initialisiert.

b) Erstellen Sie eine Funktion `printlist`, der ein Pointer auf eine Liste mit Koordinaten übergeben wird. `printlist` gibt dann alle in der Liste enthaltenen Koordinaten in der Konsole aus (z. B. “X: 2.0, Y: 4.6“)

c) Erstellen Sie eine Funktion `delcoords`, der eine Referenz auf eine Liste mit Koordinaten übergeben wird. `delcoords` löscht alle Koordinaten aus der Liste, deren x-Wert kleiner als der y-wert ist.

d) Schreiben Sie eine `main`-Funktion, in der eine Liste mit Koordinaten gefüllt wird. Die `x`-und `y`-Werte der Koordinaten sollen zufällig erstellt werden. Geben Sie die Koordinaten der Liste aus und erstellen sie ein Backup (eine zweite Liste) der ersten Liste.
Löschen Sie aus der ersten Liste alle Koordinaten, deren `x`-wert kleiner als der `y`-wert ist. Geben Sie anschließend beide Listen in der Konsole aus und überprüfen Sie, ob die richtigen Koordinaten gelöscht wurden.

Hinweis: Struct

Ein Struct verhält sich in C++ wie eine normale Klasse. Der einzige Unterschied besteht darin, dass alle Attribute automatisch auf public gesetzt sind. Die Implementierung erfolgt somit ähnlich zu der einer Klasse. Dabei wird statt dem Schlüsselwort class das Schlüsselwort struct verwendet.

```
1  struct Wertepaar{
2      int m_iX1;
3      int m_iX2;
4      Wertepaar(int x , int y){    // Konstruktor von Wertepaar
5          m_iX1 = x;
6          m_iX2= y;
7      }
8  };
```

Listing 84: Implementierung eines Structs

Hinweis/Erinnerung:

Fügen Sie `srand((unsigned)time(nullptr))` einmal pro Programm, am besten am Anfang der `main`-Funktion ein, und binden Sie die Bibliotheken `cstdlib` und `ctime` ein, um bei jeder Ausführung unterschiedliche Zufallszahlen zu erhalten.

1.22 Externe Bibliotheken

Eine (externe) Bibliothek ist eine Codesammlung, die häufig benötigte Funktionalitäten implementiert, wiederverwendbar macht und anderen Programmen zur Verfügung stellt. Mit der C++-Standardbibliothek haben Sie schon die am häufigsten verwendete aller C++-Bibliotheken kennen gelernt. Typischerweise bestehen C++ Bibliotheken aus zwei Teilen:

1. Einer Headerdatei, welche die Funktionalität der Bibliothek definiert und Programmen zur Verwendung anbietet
2. Einer kompilierten Binary-Datei, welche die Implementierung der Funktionalität (in bereits kompiliertem Maschinencode) enthält

Selbstverständlich können Bibliotheken auch aus mehreren Binary-Dateien und/oder Headerdateien bestehen.

Externe Bibliotheken sind aus mehreren Gründen bereits vorkompiliert. Zum einen verändern sich Bibliotheken nicht sehr oft, d.h. es wäre eine Zeitverschwendung die Bibliotheken bei jeder Verwendung erneut zu kompilieren. Zusätzlich verhindert eine Bibliothek in Maschinencode, dass ein Benutzer den Programmcode verändert (was bei mehrfach verwendeten Bibliotheken fatal sein könnte).

1.22.1 Statische vs. Dynamische Bibliotheken

Es gibt zwei Arten von C++ Bibliotheken: statische und dynamische Bibliotheken. Eine statische Bibliothek (oder archive) besteht aus Routinen, die in ein Programm kompiliert und gelinkt werden. Wird ein Programm mit einer statischen Bibliothek kompiliert, so wird die gesamte Funktionalität der Bibliothek Teil des Programmes.

Unter Windows ist die Erweiterung für statische Bibliotheken “.lib“ (library), unter Linux enden sie auf “.a“ (archive).

Ein Vorteil von statischen Bibliotheken (gegenüber dynamischen) ist, dass nur eine ausführbare Datei weitergegeben werden muss, um das Programm auszuführen. Die Bibliothek ist Teil des Programmes und muss nicht extra an den Benutzer des Programmes gegeben werden. Statische Bibliotheken haben den Nachteil, dass sie den Speicherbedarf eines Programmes erhöhen (da sie Teil des Programmes sind) und dass sie nur schwer aktualisierbar sind (da jedes Mal das Programm neu kompiliert werden muss). Im Rahmen dieses Praktikums wird ausschließlich mit statischen Bibliotheken gearbeitet.

Dynamische Bibliotheken bestehen aus Routinen, die während der Ausführung eines Programmes in das Programm geladen werden. Wird ein Programm mit einer dynamischen Bibliothek kompiliert, so wird die Bibliothek NICHT Teil des Programmes. Sie bleibt weiterhin eine separate Datei.

Unter Windows ist die Erweiterung für dynamische Bibliotheken “.dll“ (dynamic linked library), unter Linux enden sie auf “.so“ (shared object).

Ein Vorteil von dynamischen Bibliotheken ist, dass mehrere Programme sie (gleichzeitig) verwenden können. Man braucht nicht mehrere Kopien, wie bei statischen Bibliotheken. Ein noch größerer Vorteil ist, dass dynamische Bibliotheken auf neue Versionen aktualisiert werden können, ohne die Programme verändern zu müssen.

1.22.2 Installieren und Verwendung einer externen Bibliothek

In der Regel sollten alle für das Praktikum notwendigen Bibliotheken bereits vorinstalliert sein. Benötigt man dennoch zusätzliche Bibliotheken, müssen diese zum Beispiel mittels Befehle im Terminal installiert werden. Für dieses Praktikum sind bereits alle Libraries auf der Virtuellen Maschine vorinstalliert.

Um die Bibliothek im programmierten Programm verwenden zu können, müssen Compiler und Linker wissen, welche Bibliotheken verwendet werden sollen, wo sie sich befinden und wo die Headerdateien (hier: `curses.h`) gespeichert sind, die die Funktionalität der Bibliotheken definieren. Diese Einstellungen werden in Eclipse unter folgenden Pfad vorgenommen:

Rechtsklick auf Projekt → Properties → C/C++ General → Paths and Symbols.

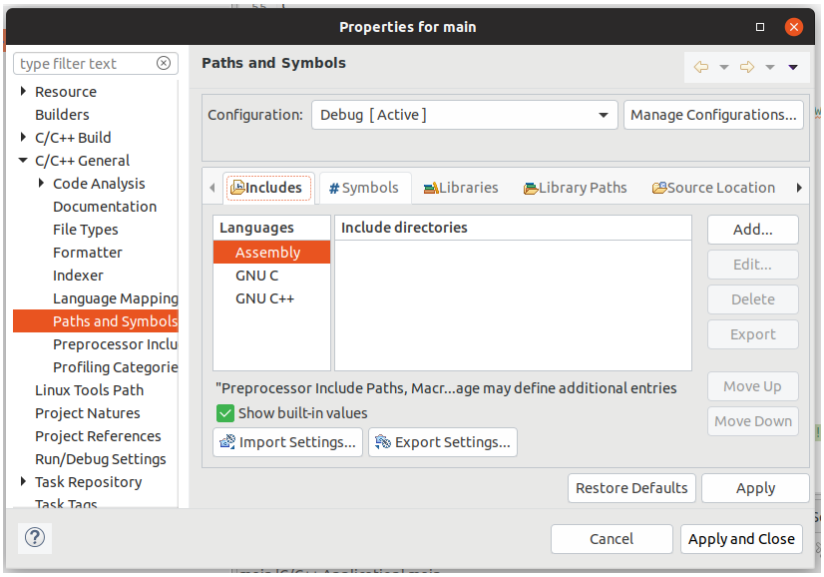


Abbildung 17: Compiler und Linker Pfade setzen

An dieser Stelle müssen im Allgemeinen drei Einstellungen und im Programmcode eine getätigt werden:

1. Dem Compiler muss mitgeteilt werden, wo sich die Headerdatei befindet, die die Funktionalität der Bibliothek definiert. In unserem bisherigen Linux Projekten wurde die Headerdatei bereits in das *include*-Verzeichnis kopiert. Ist dies nicht der Fall, so muss in der Kartei *Includes* die Option GNU C++ gewählt, dann auf Add gedrückt und im sich öffnenden Fenster der Pfad zur Headerdatei angegeben werden.
2. Dem Linker muss anschließend noch der Pfad mitgeteilt werden, an dem sich die einzubindenden Bibliotheken befinden. Im Allgemeinen geschieht dies aber in der Kartei *Library Paths* mit einem Klick auf *Add* (siehe Abbildung 18).
3. Anschließend wird im Reiter *Libraries* angegeben, welche Libraries bei der Kompilierung des Programms genutzt werden sollen. Hierzu wird der Name der Bibliothek in dem Fenster eingegeben, das sich öffnet, wenn in der Kartei *Libraries* auf *Add* gedrückt wird.

   ```
   ncurses
   ```

eingegeben werden.

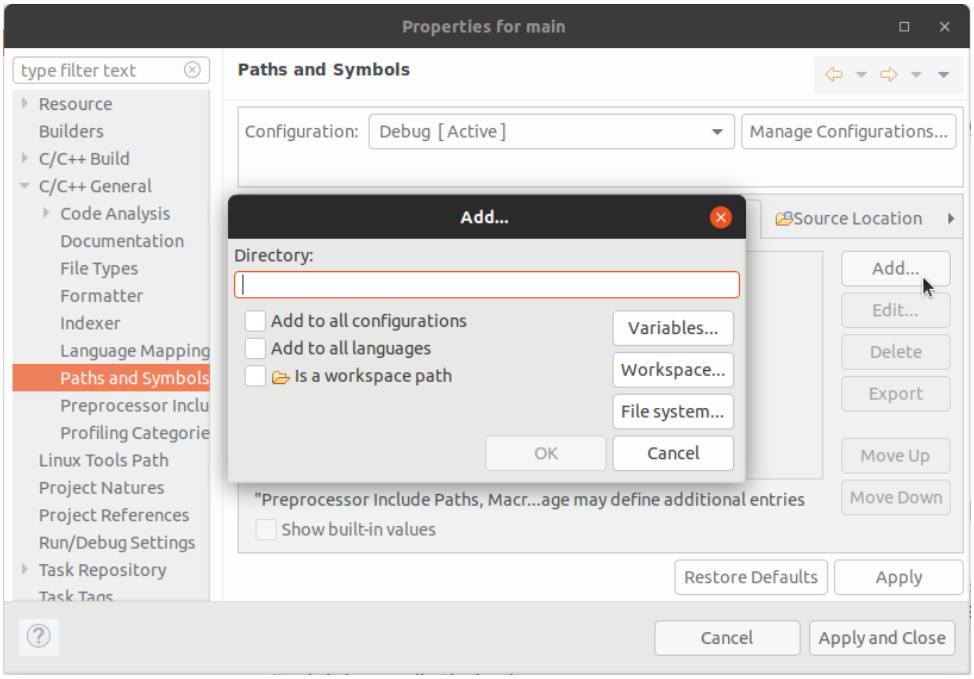


Abbildung 18: Pfade zur Headerdatei eintragen

4. Folgendes wird im **Programmcode** durchgeführt:

   Damit dem Editor die aus der externen Bibliothek verwendeten Funktionen erkennt und korrekt darstellen kann, müssen abschließend die entsprechenden Headerdateien in den Programmcode eingebunden werden. Dies geschieht mit dem Befehl `#include` am Anfang der Codedatei, in der die Funktionen genutzt werden.

Zusatzinformation für Interessierte

Dieser Kasten dient der näheren Erläuterung der vorhergehenden Aufzählung und ist weder für die folgenden Aufgaben noch für das Testat relevant):

Wird – wie in unserem Fall – C++ verwendet, sieht der GNU g++-Compiling-Befehl, den Eclipse automatisch im Hintergrund bei Verwendung des Build Buttons ausführt, im Terminal folgendermaßen aus:

g++ `-o` *test* `-I`*<Include-Pfad>* `-L`*<Library-Pfad>* `-l`*<Libraries> helloworld.cpp*

`g++`: Verwendeter Compiling-Befehl

`-o test`: Benennung des kompilierten Programms (Standard: „a.out“)

`-I<Include-Pfad>`: Angabe des Pfades für die *Include*-dateien

`-L<Library-Pfad>`: Angabe des Pfades für die Libraries

`-l<Libraries>`: Angabe der einzubindenden Libraries

`helloworld.cpp`: Angabe der zu kompilierenden Source-Dateien (cpp)

Mit den Schritten 1-3 der vorhergehenden Aufzählung wird der Compiling-Befehl um die Flags `-L`, `-I` und `-l` erweitert. Punkt vier erfolgt im Programmcode der Source-Datei (in diesem Beispiel: `helloworld.cpp`).

1.22.3 ncurses

ncurses ist eine Bibliothek, die das Erstellen von graphischen Benutzerschnittstellen in der Konsole ermöglicht (d.h. man kann ganze Fenster und Dialoge in der Konsole erstellen). Für die Steuerung des Roboters/der Simulation, die in den nächsten Tagen implementiert wird, sind aber nur wenige und periphere Funktionen von ncurses notwendig.

Wenn man z.B. den Roboter mit der Tastatur steuern möchte, so soll der Roboter direkt auf die Eingaben reagieren und soll nicht auf eine Bestätigung durch die Entertaste (wie bei `std::cin` notwendig) warten.

1.22.3.1 Verwendung von ncurses

Die Verwendung von ncurses geschieht in drei Schritten.

1. Vor der Verwendung muss ncurses mit den gewünschten Funktionen eingeschalten werden. Hierfür gibt es folgende Methoden:
   - `initscr();` // Initialisiert das Standardfenster für ncurses
   - `nodelay(stdscr, TRUE);` // Eingaben werden sofort verarbeitet.
   - `noecho();` // Eingaben werden nicht angezeigt
2. Nach Abschluss der Wendung muss ncurses ausgeschalten werden. Dies geschieht durch einen Aufruf von:
   - `endwin();` // beendet ncurses und kehrt zur Standardeingabe zurück.
3. Die eigentliche Verwendung geschieht über folgende drei Funktionen:
   - `int getch();` //Liest ein Zeichen aus der Konsole als ASCII. Wird kein Zeichen eingegeben so gibt getch() -1 zurück (Kann auch direkt als char verwendet werden, siehe Listing 85).
   - `clear();` // Löscht alle Ausgaben in der Konsole
   - `printw("abc, %f", fnum)` // Gibt einen Text in der Konsole aus

Die Funktion `printw()` funktioniert wie die aus C bekannte Funktion `printf()`. Der auszugebende Text wird dabei als C-string übergeben. Parameter werden über *%Parametertyp* in den String eingefügt und dann mit Komma getrennt und in der richtigen Reihenfolge nachgestellt. Gängige Parameter sind %f = float, %d = int, %c= char, %s = string.

Solange *ncurses* aktiv ist, kann die Standardeingabe/Ausgabe nicht verwendet werden (d.h. `std::cin`, `std::cout`, `std::endl` etc. funktionieren nicht wie gewünscht). Das folgende Listing 85 verdeutlicht die Verwendung:

```
1   #include "ncurses.h"
2
3   int main()
4   {
5       // Irgendwelcher Code der mit der Standardausgabe (cin, cout) arbeitet.
6       int iNumber = 0;
7       double dNumber = 0.0;
8       char chValue = ‘A’;
9       std::string sValue = "Ich bin ein String";
10
11      // Umschalten auf ncurses und Initialisierung
12      initscr();
13      nodelay(stdscr,TRUE);
14      noecho();
15
16      while ( /* Bedingung */ )
17      {
18          iNumber = getch();                      // Einlesen eines Zeichens
19
20          if (iNumber != -1)                      // stellt sicher das ein Zeichen eingegeben wurde.
21          {
22              // Verarbeitung der Eingabe
23              // …irgendwelcher Code
24
25              clear();                            // Löschen aller Ausgaben in der Konsole
26              // Folgende Zeilen geben eine Textausgabe in der Konsole aus
27              printw("Eingabe ist: %i", iNumber);   // Ausgabe: integer (Alternativ: %d)
28              printw("Eingabe ist: %f", dNumber);   // Ausgabe: double
29              printw("Eingabe ist: %c", chValue);   // Ausgabe: char
30              printw("Eingabe ist: %s", sValue);    // Ausgabe: string
31          }
32      }
33      endwin();                                   // Zurückschalten auf Standardausgabe (cout)
34  }
```

Listing 85: Verwendung von ncurses

Besondere Aufmerksamkeit ist der Zeile 18 zu widmen. Da `getch` in einer `while`-Schleife aufgerufen wird, kann der Aufruf der Methode sehr oft erfolgen (eventuell mehrere Hundert Mal pro Sekunde). Bei den meisten dieser Aufrufe wird der Benutzer keine Taste drücken. Würden aber bei jedem Aufruf `clear` und `printw` aufgerufen werden, so würde die Bildschirmausgabe ständig gelöscht und neu geschrieben werden. Dies würde in einem unlesbaren Flimmern des Bildschirms enden. Aus diesem Grund wird in Zeile 20 über die `if`-Abfrage sichergestellt, dass nur Werte von `getch` verarbeitet werden, die eine Eingabe des Benutzers repräsentieren.

1.22.3.2 Einbindung von ncurses

Um die externe Bibliothek ncurses in der nächsten Aufgabe verwenden zu können, müssen Schritte drei und vier aus Kapitel 1.22.3.2 ausgeführt werden (Die ersten beiden werden erst in den Aufgaben von Tag sechs wichtig):

3. Im Reiter Libraries muss die Bibliothek *ncurses* eingebunden werden, indem

   ```
   ncurses
   ```

   eingegeben wird.

4. Für die Nutzung von ncurses muss hier


```
#include "ncurses.h"
```

angegeben werden.

Aufgabe: ncurses

a) Erstellen Sie ein neues Projekt in Eclipse und binden sie ncurses in dieses Projekt ein (siehe Kapitel 1.22.3.2).

b) Erstellen Sie ein Programm, das über ncurses eine Interaktion mit dem Benutzer durchführt. Der Benutzer gibt in einer Schleife mehrere Zeichen ein, die in einer doppelt verketteten Liste gespeichert werden. Wenn der Benutzer ein Zeichen eingibt, so werden die bisherigen Ausgaben in der Konsole gelöscht, das Zeichen wird in der Liste gespeichert und dann auf der Konsole angezeigt. Gibt der Benutzer kein Zeichen ein, so geschieht nichts. Die Eingabe (und damit ncurses) wird bei der Eingabe von 'q' beendet und die Liste wird von hinten nach vorne in der Standardausgabe ausgegeben.

Hinweis: Ausführung von ncurses

ncurses ist für die Ausführung in einem Linux-Terminal entwickelt worden und nicht kompatibel mit der in Eclipse eingebetteten Konsole. Dies bedeutet, dass Programme, die ncurses verwenden, nicht aus Eclipse gestartet werden können.

Programme mit ncurses lassen sich folgendermaßen ausführen

1) Starten Sie ein Linux-Terminal, z.B. über das Konsole-Icon auf dem Desktop oder über Applications → Utilities → Terminal im Hauptmenü. In diesem Terminal lassen sich nun Befehle ausführen und Programme aufrufen.

2) Manövrieren sie über den Linux-Befehl `cd` in das Verzeichnis, in dem sich ihre ausführbare Datei befindet. Beispiel:
```
cd /home/pi/folder1/folder2
```

3) Sie können sich den Inhalt eines Verzeichnisses über den Befehl `ls` anzeigen lassen.

4) Befinden sie sich im richtigen Verzeichnis (im Debug Ordner des jeweiligen Projekts), so können Sie das Programm mit folgendem Befehl ausführen:
```
./Programmname
```

Hinweis: Nutzung von printw()

Für die Ausgabe von Variablen mittels `printw()` siehe Listing 85.

2 Tag 4

2.1 Einleitung

Die Hardware stellt die zentrale Komponente des Roboters dar (Übersicht siehe Abb. 18). Der Roboter „Forbot A4“ wurde von der Firma Roboterwerk entwickelt und produziert. Er ist als Plattform für die Automatisierungstechnik konzipiert. Mit dessen Hilfe lassen sich die verschiedensten Aufgaben und Problemstellungen erforschen sowie vermitteln. Aus diesem Grund ist er auch für die Lehre sehr gut geeignet.

2.2 Hardwareaufbau

Das Edelstahlgehäuse mit Einbauten aus Aluminium verleiht dem Roboter eine große Widerstandsfähigkeit gegenüber mechanischer Beanspruchung. Die Steuerung ist in die Bodenwanne eingebaut und wird über einen Akkupack oder ein Netzteil mit Strom versorgt.

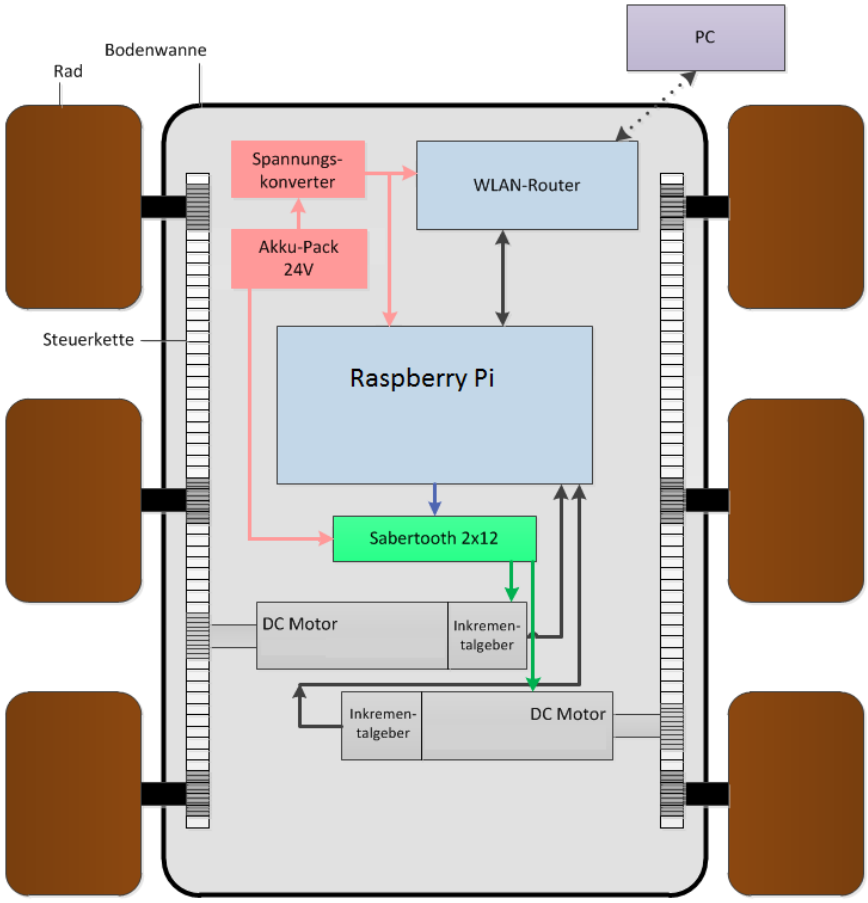


Abbildung 19: Hardware-Übersicht

Linke Seite:
Index [1]

Rechte Seite:
Index [0]

Gesteuert werden die Motoren über einen Motorcontroller (Sabertooth), der ein Servo-Signal vom Raspberry Pi erhält. Die Drehzahl des Motors wird mit Hilfe eines Planetengetriebes untersetzt. Zur Kraftübertragung auf die Räder wird ein Kettentrieb verwendet. Dieser treibt jeweils auf einer Seite drei Räder synchron an. Die Drehzahl

wird mit Hilfe der in den Gleichstrommotoren integrierten Inkrementaldrehgebern überwacht, die an den Raspberry Pi angeschlossen werden.

2.2.1 Kommunikation

Die Kommunikation mit externen Geräten erfolgt über einen WLAN-Router auf dem Gehäuse des Roboters. Die kabellose Verbindung erhöht die Flexibilität des Roboters. Er kann sich frei bewegen und ist nicht auf eine physikalische Verbindung zu einem PC oder sonstigen Geräten angewiesen. Auf den Raspberry Pi kann entweder über einen Remote-Desktop-Zugriff oder via SSH-Verbindung zugegriffen werden.

Abbildung 20: Raspberry Pi

2.2.2 Steuerung

Gesteuert wird der Roboter über einen Raspberry Pi (siehe Abb. 19). Der Raspberry Pi ist ein kreditkartengroßer Einplatinencomputer, der von der britischen *Raspberry Pi Foundation* entwickelt wurde. Er ist im Vergleich zu üblichen PCs sehr einfach aufgebaut und wurde mit dem Ziel entwickelt, Menschen den Erwerb von Programmier- und Hardwarekenntnissen zu erleichtern. Die Platine enthält ein Ein-Chip-System von Broadcom mit einem 700-MHz-ARM11-Prozessor und 512 MB Arbeitsspeicher. Außerdem hat er eine Ethernet-Schnittstelle und zwei USB-Anschlüsse.

Als Betriebssystem können angepasste Linux-Versionen oder andere Betriebssysteme installiert werden, welche die ARM-Architektur unterstützen. Eine eigene Festplatten-Schnittstelle ist nicht vorhanden, stattdessen können SD-Speicherkarten als Bootmedium benutzt werden.

Bis November 2013 wurden mehr als 2 Millionen Geräte verkauft. Der Raspberry Pi erhielt mehrere Innovationspreise, mittlerweile existiert ein großes Zubehör- und Softwareangebot für verschiedenste Anwendungsbereiche. Der Raspberry Pi stellt die Hauptzentrale des Roboters dar. Ohne sie würde der Roboter nicht funktionieren.

2.2.3 Antrieb

Der Antrieb wird mit Hilfe zweier Gleichstrommotoren realisiert. Jeweils ein Motor treibt über eine Steuerkette eine Seite des Roboters mit je drei Rädern an. Diese Anordnung ermöglicht dem Roboter eine hohe Traktion sowie Flexibilität in der Steuerung. Die Lagerung der Räder ist starr ausgelegt und direkt in die Bodenwanne integriert.

Die Motoren können nicht direkt über den Raspberry Pi angesteuert werden, da die Ausgänge des Raspberry Pi nur eine sehr begrenzte elektrische Leistung bereitstellen können. Das Steuersignal muss deshalb mittels der Leistungselektronik des Motorcontrollers umgewandelt werden. Im Roboter wird dafür ein Sabertooth 2X12 verwendet. Dieser verstärkt lediglich die Amplitude des Eingangssignals, der zeitliche Verlauf bleibt dabei unverändert. Der Sabertooth erhält vom Raspberry Pi ein Servo-Signal, interpretiert es und schaltet schließlich das verstärkte Signal auf die Motoren.

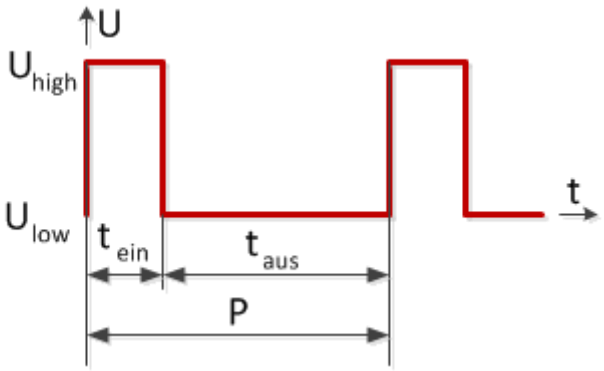


Abbildung 21: PWM-Signal

2.2.4 Pulsweitenmodulation

In Abbildung 20 ist ein PWM-Signal zu sehen. Es hat die Periodendauer P. Diese wird als Zykluszeit des PWM-Signals bezeichnet. Für eine bestimmte Zeit t_{ein} ist das Signal „High“ und für die Zeit $t_{aus} = P - t_{ein}$ „Low“. t_{ein} wird als Impulsdauer bezeichnet.

Das Verhältnis $\frac{t_{ein}}{t_{ein}+t_{aus}} = \frac{t_{ein}}{P}$ wird Tastverhältnis genannt. Multipliziert mit 100 beschreibt es in Prozent, wie lange innerhalb einer Periode das Signal „High“ ist.

Einerseits kann ein PWM-Signal zur Signalübertragung verwendet werden, andererseits für die Leistungssteuerung

2.2.4.1 Einsatz als Steuersignal

Wird das PWM-Signal als Steuersignal eingesetzt, wird häufig der RC-Standard (Radio Control) gewählt. Die Interpretation erfolgt hierbei ausschließlich über die Einschaltdauer t_{ein}. Dabei bedeutet 1,5 ms Einschaltzeit die Ruheposition, 1 ms die linke Endposition und 2 ms die rechte Endposition. Die Periodendauer spielt keine Rolle. Häufig wird eine Periodendauer von 20ms verwendet. Es handelt sich also um ein Servo-Signal. Auf den Roboter angewendet bedeutet eine Einschaltdauer von 2 ms, dass der Roboter mit maximaler Geschwindigkeit vorwärts fährt. Dementsprechend bedeuten 1 ms volle Rückwärtsfahrt. In der Praxis können kleinere Abweichungen bezüglich dieser Werte auftreten.

2.2.5 Inkrementalgeber

Auf den DC-Motoren ist ein Inkrementalgeber angeflanscht. Er arbeitet mit Hall-Sensoren. Ein Inkrementalgeber ist ein System zur Erfassung einer Dreh- oder Längsbewegung. Dabei handelt es sich um ein relatives Maßsystem. Es ist nicht möglich, die absolute Position zu bestimmen, sondern nur die Position bezüglich eines Referenzpunktes. Bei Neustart des Systems oder bei Start eines neuen Programms ist ein Referenzieren nötig, um damit Abstände bestimmen zu können. Der Inkrementalgeber liefert einen Gray-Code (siehe Abb. 4). Er besitzt den Vorteil, dass sich in binärer Schreibweise bei jedem Impuls nur ein Bit ändert. Damit ist eine erhöhte Übertragungssicherheit gegeben.

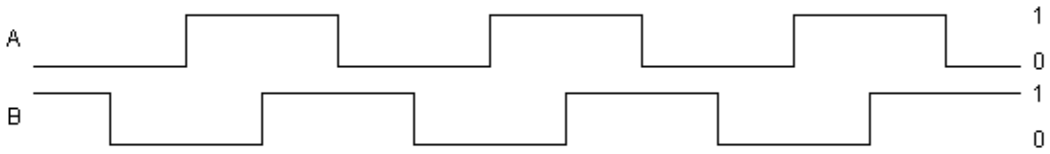


Abbildung 4: Signal eines 2-Kanal-Impulsgebers

2.2.5.1 Prinzipielle Funktionsweise

Auf die Abtriebswelle eines Motors wird eine Lochscheibe montiert. Diese wird mit einer Hintergrundbeleuchtung bestrahlt. Sobald sich die Welle in Bewegung versetzt, gelangt die Hintergrundbeleuchtung nicht mehr kontinuierlich auf den Lichtsensor. Dadurch wird ein binäres Signal erzeugt. Wenn die Anzahl der Löcher der Lochscheibe bekannt ist, kann durch einfaches Zählen der Impulse der Drehwinkel oder die Geschwindigkeit bestimmt werden.

Die Erzeugung des Signals kann auf unterschiedliche Weise erfolgen. Häufig wird ein optisches Verfahren verwendet. Es existieren aber auch magnetische (Hall-Effekt) und mechanische Verfahren. Im verwendeten Roboter ist der Inkrementalgeber mit Hilfe eines Hall-Sensors realisiert. Dieser ermöglicht die berührungslose Generierung des binären Signals. Der Hall-Effekt stellt einen Zusammenhang zwischen Spannung, Stromstärke, magnetischer Felddichte, Plattendicke und der Hall-Konstante, dar.

Bei dem verwendeten Inkrementalgeber handelt sich um einen 2-Kanal Impulsgeber. Dieser liefert zwei Kanäle (Spur A, Spur B), deren Impulse um jeweils 90° versetzt sind (siehe Abb. 4). Bei beiden handelt es sich wiederum um binäre Signale. Es lässt sich nicht nur die Geschwindigkeit bzw. die Anzahl der Impulse feststellen, sondern auch die Richtung der Bewegung. Beide Kanäle müssen separat an die Eingänge des Raspberry Pi angeschlossen werden.

2.2.5.2 Geschwindigkeitsbestimmung

Über Interrupts können die Flankenwechsel (im folgenden Impulse genannt) im Signal der Inkrementalgeber erfasst werden. Werden diese dem Richtungssinn entsprechend gezählt (vorwärts: +; rückwärts: -), lässt sich die Geschwindigkeit ermitteln. Dazu wird die Anzahl der Impulse mit fester Periode abgefragt und über den Radumfang sowie die Anzahl von Impulsen in einer Umdrehung umgerechnet. Der Radumfang beträgt 27cm und eine Umdrehung des Rades erzeugt 1567 Impulse (Flankenwechsel in beiden Signalen).

$$Geschwindigkeit = \frac{gemessene\ Impulse \times Radumfang}{1567\ \frac{Impulse}{Umdrehung} \times Periode}$$

2.2.6 Stromversorgung

Die Strom- bzw. Spannungsversorgung übernimmt ein Netzteil oder Akkupack der 2,5 Ah bei 24 V liefert. Dieser ist in der Bodenwanne des Roboters eingebaut. Die Hauptversorgungspannung liegt somit bei 24 V. Der Sabertooth sowie Motoren werden damit versorgt. Ein DC-Wandler liefert die nötige Spannung für den WLAN-Router und einen zweiten Spannungswandler, der den Raspberry Pi über Mikro-USB mit 5V versorgt.

2.3 Programmierung der Steuerung

Im Folgenden wird eine Steuerung für den Roboter Forbot A4 bzw. für dessen Simulation mit der Tastatur entwickelt. Im ersten Schritt wird eine einfache Steuerung programmiert, die anschließend im zweiten Schritt um eine Geschwindigkeitsregelung erweitert wird.

2.4 Steuerungsaufbau

Der Roboter wird durch die Klasse **KeyboardControl** mit der Tastatur gesteuert. Diese Klasse koordiniert die Aktionen des Roboters anhand von Tastatureingaben. Die Klasse hat folgenden Aufbau:

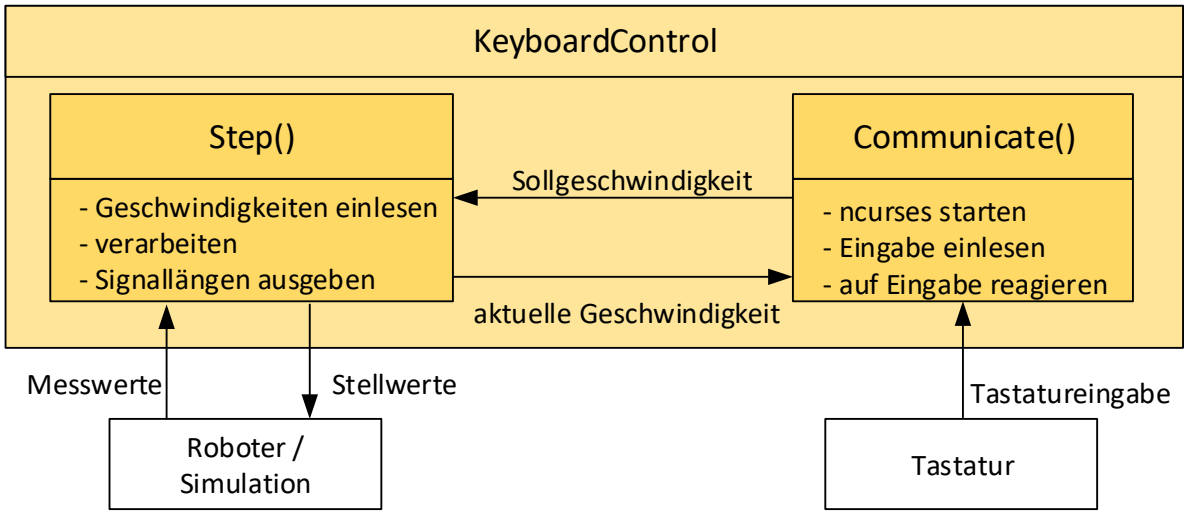


Abbildung 22: Aufbau der Klasse `KeyboardControl`

Die Methode `Communicate()` kommuniziert mit dem Benutzer bzw. mit der Tastatur. `Communicate()` liest einzelne Zeichen des Benutzers ein und stellt daraufhin die Wunschgeschwindigkeit des Roboters ein.

Gleichzeitig organisiert die Methode `Step()` die Kommunikation mit dem Roboter/der Simulation. `Step()` nimmt den Wert der Soll-Geschwindigkeit, der über die Methode `Communicate()` eingestellt wurde, und verarbeitet diesen. Dazu wird der aktuelle Messwert der Geschwindigkeit eingelesen, ausgewertet und anschließend gegebenenfalls eine neue Stellgröße an den Roboter gesendet.

Wie in der Grafik zu sehen, findet zusätzlich ein Austausch von Informationen zwischen `Step()` und `Communicate()` statt. Dieser wird durch zwei Arrays gewährleistet; je eines für die Übergabe der aktuellen Geschwindigkeit und der Wunschgeschwindigkeit. Die Arrays speichern an der Stelle ‘0‘ die Geschwindigkeit des rechten und an der Stelle ‘1‘ die Geschwindigkeit des linken Motors.

Aufgabe: Erstellen der Klasse KeyboardControl

Erstellen Sie eine Klasse `KeyboardControl`. Diese soll zunächst eine (noch leere) Methode `Communicate()`, eine (noch leere) Methode `Step()` und die beiden Arrays (`double`) für Soll- und Ist-Geschwindigkeit enthalten.

2.5 Die Methode Communicate()

Damit nach Tastatureingaben keine Bestätigung mit der Enter-Taste notwendig ist, soll die Bibliothek `ncurses` verwendet werden.
Dazu muss die externe Bibliothek in den Projekteigenschaften eingebunden werden (siehe Abschnitt 1.22). Die `Communicate()`-Funktion soll den in Abbildung 23 dargestellten Aufbau aufweisen.

In der ersten Schleife findet die aktive Kommunikation zwischen Programm und Benutzer statt. Die Eingaben (‘w‘, ‘a‘, ‘s‘, ‘d‘, ‘q‘ und ‘b‘) sollen verarbeitet werden. Durch Drücken der Taste ‘w‘ (vorwärts) bzw. ‘s‘ (rückwärts) soll die aktuelle Wunschgeschwindigkeit der Motoren jeweils um 0,01 erhöht bzw. vermindert werden. Durch Drücken der Taste ‘a‘ soll die aktuelle Wunschgeschwindigkeit des rechten Motors um 0,005 erhöht und die des linken Motors um 0,005 vermindert werden. Die Ansteuerung der Motoren durch Drücken der Taste ‘d‘ erfolgt entsprechend entgegengesetzt.

Hinweis: Begrenzung der Sollgeschwindigkeiten

Während der Ansteuerung des Roboters muss sichergestellt werden, dass die Geschwindigkeiten der linken und rechten Räder das Intervall [-0.5, +0.5] m/s nicht über- bzw. unterschreiten!

Beim Drücken der Tasten ‘b‘ („*break*“) und ‘q‘ („*quit*“) soll der Roboter anhalten. Im Gegensatz zu *break*, soll der Roboter bei *quit* die erste Schleife verlassen und die Tastatursteuerung beenden. Voraussetzung für das erfolgreiche Beenden des Programms ist die Sicherstellung, dass der Roboter nicht mehr fährt (d.h. wenn die Ist-Geschwindigkeiten des Roboters gleich null sind). Dies wird in der zweiten Schleife überprüft. Anschließend kann `ncurses` ausgeschaltet und das Programm verlassen werden.

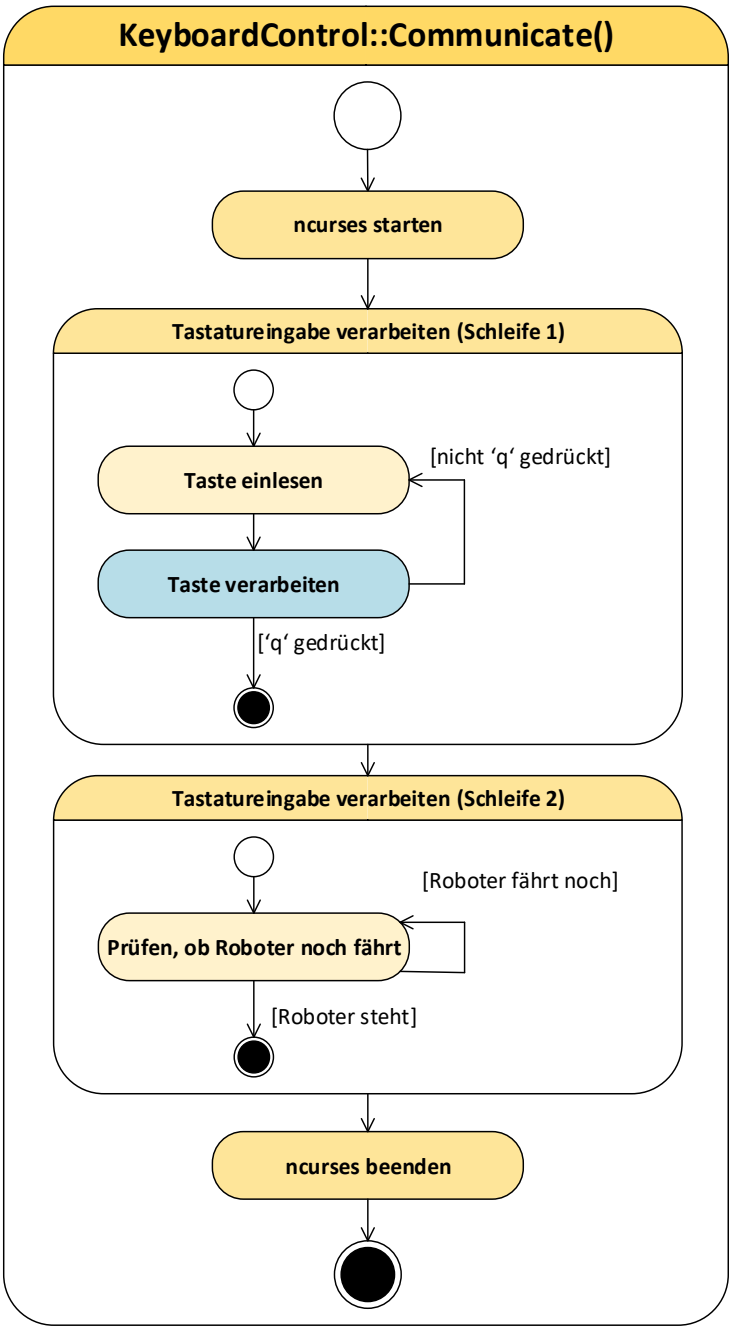


Abbildung 23: Aufbau der Methode `Communicate()`

Aufgabe: Erstellen der Communicate-Methode

a) Erstellen Sie die Methode `Communicate()`. Folgen Sie dabei der oben angegebenen Beschreibung und benutzen Sie die angegebenen Werte.

b) Legen sie die Datei main.cpp an und definieren Sie darin die Funktion `main`. Testen Sie damit die Klasse `KeyboardControl` und die Methode `Communicate()`. Die Sollgeschwindigkeit, die durch den Benutzer eingestellt wird sowie die letzte Taste, die gedrückt wurde, sollen in der Konsole angezeigt werden.

Hinweis: getch()

Achten Sie darauf, dass ein Zeichen nur dann ausgewertet wird, wenn auch wirklich eines eingelesen wurde. Nutzen Sie dazu die Eigenschaft der `getch()`-Methode. Sollten Sie für die Geschwindigkeitsbegrenzung den Betrag benutzen, müssen Sie für `floats` die Funktion `fabs(…)` aus `<cmath>` benutzen.

2.6 Die Step-Funktion

Die `Step()`-Funktion stellt die Schnittstelle zwischen der `Communicate()`-Funktion und dem Roboter bzw. der Simulation dar. Sie folgt dem Schema in Abbildung 24. Zuerst soll die Ist-Geschwindigkeit des Roboters eingelesen und in einem Array gespeichert werden. Danach soll aus der Wunschgeschwindigkeit ein Signal generiert werden, welches an den Roboter gesendet wird. Die Simulation (und auch der Roboter) akzeptieren als Eingangssignal Signallängen zwischen [1000 µs, 2000 µs]. Die Nulllage des Roboters, bei der keine Bewegung stattfindet, liegt bei 1500 µs. Die Geschwindigkeiten zwischen [-0.5, 0.5] m/s müssen somit in das Intervall [1000 µs, 2000 µs] umgerechnet werden. Für spätere Anwendungen (geregeltes System) muss der Wertebereich der Stellgrößen [1000 µs, 2000 µs] mit Hilfe einer einfach `if`-Anweisung überprüft werden.

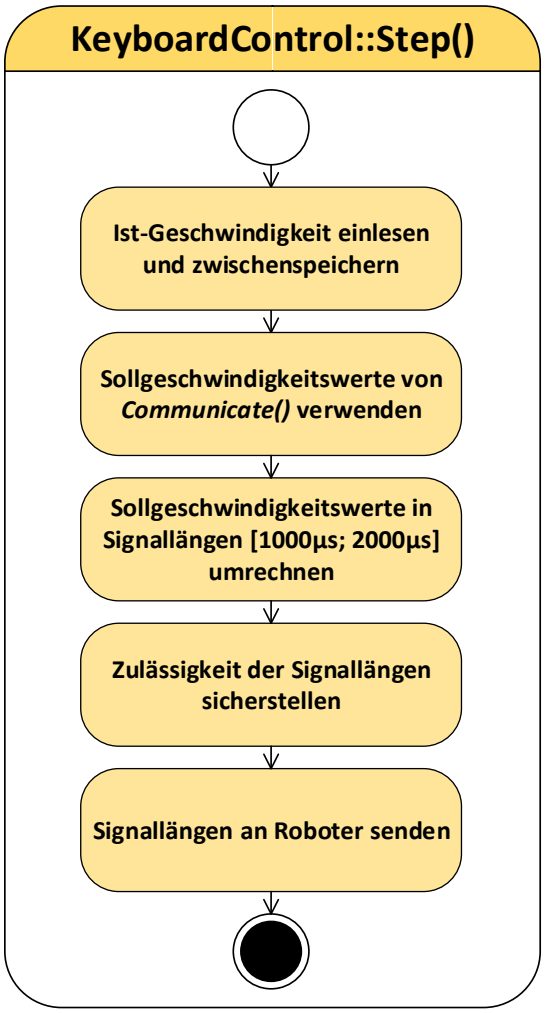


Abbildung 24: Schema der Methode `Step()`

Das Auslesen der aktuellen Geschwindigkeiten der Räder sowie das Senden der Stellgeschwindigkeit an den Motorcontroller erfolgt über die Schnittstelle **InterfaceSIM,** die Sie auf dem Laufwerk `sefi (S:)` finden (`InterfaceSIM.h` und `InterfaceSIM.cpp`). Der Ablauf folgt dem Schema aus Abbildung 26.

Die Simulation befüllt die Datei *actualValue* mit den momentanen Ist-Geschwindigkeiten des simulierten Rovers und liest gleichzeitig die neu umzusetzenden Geschwindigkeiten aus der Datei *targetValue* aus und setzt diese um. Das Interface führt die gleichen Operationen auf Seiten des Programmcodes in umgekehrter Richtung aus (*targetValue* wird beschrieben und *actualValue* wird ausgelesen). Den Befehl zum Beschreiben und Auslesen der Dateien gibt der von Ihnen geschriebene Programmcode durch die Funktionen `setOutputs()` und `getInput()` (näheres dazu in der folgenden Aufzählung).

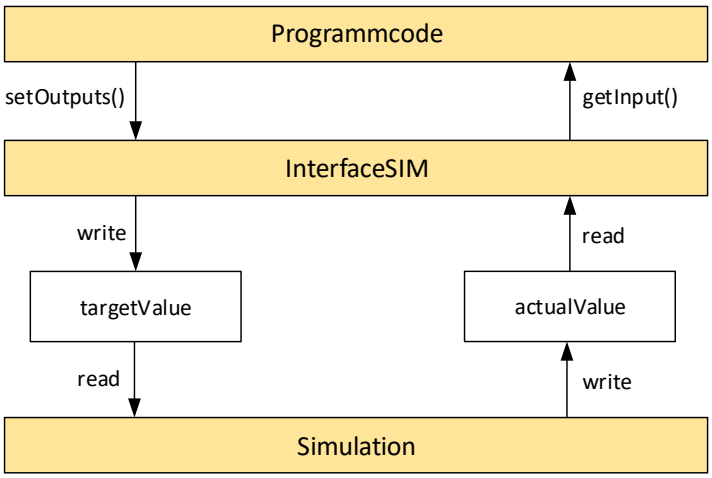


Abbildung 25: Funktionsweise der Klasse `InterfaceSIM`

Für die Einbindung und Nutzung des Interfaces müssen Sie folgende Schritte umsetzen:

1. Kopieren Sie diese beiden Dateien in das Projekt.
2. Erstellen Sie ein Objekt der Schnittstellenklasse in der Klasse **KeyboardControl** .
3. Initialisieren Sie dieses mit der Methode

   ```
   void Initialize(double Zeitschrittlänge, void
        (*pFunction)(void));
   ```

   Als Übergabeparameter wird die Länge der Zeitschritte in [s], in denen die Methode `Step()` aufgerufen wird, sowie die statische Transferfunktion (siehe 2.7 **Parallelisierung der Ausführung**) benötigt. Die Zeitschrittlänge beträgt in unserem Fall 0,04s und entspricht der Zykluszeit eines Simulationsschrittes. Achten Sie auf die korrekte Syntax zum Aufrufen einer Methode einer Klasse mit Hilfe eines Objekts.

   Diese Methode wird für die Initialisierung des zur `Communicate()` automatischen, parallelen Aufruf von `Step()` benötigt (siehe 2.7 **Parallelisierung der Ausführung**).

4. Für das Auslesen der aktuellen Geschwindigkeiten stellt die Schnittstelle die Methode

   ```
   double* GetInput();
   ```

   zur Verfügung. Sie gibt einen Pointer zurück, der auf ein `double`-Array mit zwei Elementen zeigt. Das erste Element enthält die aktuell gemessene Geschwindigkeit [m/s] der rechten Seite und das zweite Element die aktuell gemessene Geschwindigkeit [m/s] der linken Seite des Roboters. Die Werte für die gemessenen Geschwindigkeiten können mit einem Pointer (`pdInput`) in das bereits angelegte Array für Ist-Geschwindigkeit gespeichert werden.

Hinweis: Aufrufen von GetInput()

Die Methode `GetInput()` darf nur einmal zu Beginn der `Step`-Methode aufgerufen werden, da der Zeitschritt, mit dem die `Step`-Methode ausgeführt wird, in die Berechnung der Geschwindigkeit eingeht. Achten Sie außerdem darauf, nicht den Adressraum des Arrays zu verlassen.

5. Das Setzen der Sollwerte ermöglicht die Schnittstelle mit der Methode

   ```
   void SetOutputs(int* iMicros).
   ```

   Dazu muss ein Pointer auf ein `int`-Array mit zwei Elementen, welche die Signallängen [µs] enthalten (1. Element: rechter Motor; 2.Element linker Motor), übergeben werden. Achten Sie darauf, dass die Signallängen nur zwischen 1000 µs und 2000 µs liegen dürfen. Initialisieren Sie das Speicherarray im Konstruktor von `KeyboardControl` mit der Nulllage der Signallänge (1500µs). Sie können das Array der Funktion ohne Nutzung eines weiteren Pointers übergeben (mit `setOutpus(iMicros)`).

6. Für die Simulation müssen außerdem die externen Bibliotheken `<rt>` *und* `<jsoncpp>` eingebunden werden. Dazu müssen in der Kartei *Libraries* die folgenden Einträge zusätzlich hinzugefügt werden (vgl. 1.22.3.2)

   ```
   rt
   ```

   ```
   jsoncpp
   ```

(Beachten Sie, dass der ncurses Ordner unter *Library Paths* hinterlegt sein muss.)

Hinweis: Einbindungen von externen Bibliotheken

`#include <rt>` und `#include <jsoncpp>` werden **nicht** benötigt.

Sie benötigen die Einbindung als Library.

Aufgabe: Einbinden der Schnittstelle und Erstellen der Step-Methode

a) Binden Sie die Schnittstelle zur Simulation wie oben beschrieben in das Projekt ein. Legen Sie ein Objekt der Schnittstelle in der Klasse `KeyboardControl` an und initialisieren sie dieses im Konstruktor. Das zweite Argument der Initialisierungsfunktion lassen Sie leer, dieser wird im nächsten Schritt nachgetragen.

b) Implementieren Sie die Methode `Step()`.

2.7 Parallelisierung der Ausführung

Prinzipiell ist die Tastatursteuerung des Roboters bzw. der Simulation an dieser Stelle funktionsfähig. Allerdings ist es zu diesem Zeitpunkt noch nicht möglich die `Step()`-Funktion und die `Communicate()`-Funktion parallel auszuführen. Dieses ist jedoch notwendig, um gleichzeitig mit dem Benutzer und dem Roboter zu interagieren. Das Parallelisieren der Funktionen `Step()` und `Communicate()` erfolgt mit Hilfe der Schnittstelle.

Dazu muss der Schnittstelle ein Pointer auf die `Step()`-Funktion der Klasse `KeyboardControl` übergeben werden. Dies wird in den folgenden 5 Schritten umgesetzt:

1. Fügen Sie der Klasse `KeyboardControl` einen von außen sichtbaren, statischen Pointer `transferPointer` auf ein Objekt von `KeyboardControl` hinzu.
2. In der .cpp Datei von `KeyboardControl` muss der Pointer mit folgender Zeile referenziert werden:

   ```
   KeyboardControl* KeyboardControl::transferPointer;
   ```

3. Im Konstruktor der Klasse `KeyboardControl` wird der Pointer anschließend auf den Wert `this` gesetzt. Damit zeigt der Pointer nun auf die Klasse selbst. Dies sollte ganz am Ende des Konstruktors geschehen.
4. Fügen Sie der Klasse `KeyboardControl` eine von außen sichtbare, statische Funktion `transferFunction` hinzu, die weder Übergabeparameter nimmt, noch Werte zurückgibt.
5. In der `transferFunction` soll die `Step()`-Funktion über den `transferPointer` aufgerufen werden. Dies geschieht mit dem Befehl:

   ```
   transferPointer->Step();
   ```

Abschließend muss die parallele Ausführung der `Step()`-Funktion noch gestartet, bzw. am Ende des Programmes wieder beendet werden. Beides geschieht in der Methode `Communicate()`. Zum Starten muss die Zeile

```
1  sigprocmask(SIG_UNBLOCK, &<NameDesSchnittstellenObjekts>.mask, nullptr);
```

eingefügt werden. Führen Sie diese Zeile aus, bevor die Kommunikation mit dem Benutzer startet. Zum Beenden wird die Zeile


```
1  sigprocmask(SIG_BLOCK, &<NameDesSchnittstellenObjekts>.mask, nullptr);
```

verwendet. Diese Zeile muss nach Abschluss jeglicher Kommunikation mit dem Roboter eingefügt werden. Beachten Sie den Hinweis.

Durch die erste Methode wird die Step()-Funktion zyklisch aufgerufen und durch die Zweite das zyklische Aufrufen wieder beendet.

Aufgabe: Parallelisierung der Ausführung

a) Vervollständigen Sie die Initialisierung von `InterfaceSIM`.

b) Führen Sie die Parallelisierung der Ausführung wie oben beschrieben durch.

c) Kompilieren Sie das Programm.

Hinweis: <NameDesSchnittstellenObjekts>

Ersetzen Sie <NameDesSchnittstellenObjekts> durch den Namen Ihres Objekts der Klasse `InterfaceSIM`.

2.8 Ausführen des Programms

Das Programm kann nicht in Eclipse, sondern nur in der Konsole ausgeführt werden. Zunächst muss allerdings die Simulation gestartet werden. Sie finden die Simulation „MoonRover.x86_64“ als Verknüpfung auf dem Schreibtisch.

Starten Sie die Simulation, um Ihr Programm zu testen. Wenn sie das Feld „Evaluation Mode“ aktivieren, können Sie die gefahrene Route des Roboters nachverfolgen. Mit den Tasten „1“, „2“, „3“ und „4“ wechseln Sie zwischen den vorkonfigurierten Kameraperspektiven.

Um Ihr Programm auszuführen, müssen Sie mit dem Terminal in den Workspace-Ordner wechseln, in dem ihr Code gespeichert ist. Dieser Befindet sich auf Ihrem Netzlaufwerk. Wenn Sie folgenden Befehl in der Konsole eingeben, wechseln Sie in Ihren Projektordner:

```
cd /mnt/hgfs/Netzlaufwerk/Workspace/<Projektname>/Debug
```

Das `cd` steht dabei für **change directory.** Mit

```
ls
```

wird alles angezeigt was sich in dem Ordner befindet, in dem Sie sich gerade im Terminal aufhalten. Der Befehl

```
pwd
```

zeigt Ihnen den momentanen Pfad an.

Um die Datei auszuführen, muss

```
./<Projektname>
```

in die Konsole eingegeben werden. Dabei sind die Angaben in eckigen Klammern Platzhalter, die ohne die Klammern einzugeben sind. Der Befehl `cd XY/` wechseln Sie mit der Eingabeaufforderung in den Unterordner `XY`. Mit dem Befehl `cd ..` können

Sie wieder eine Hierarchieebene nach oben wechseln. Der Befehl `ls` zeigt Ihnen alle Dateien und Ordner innerhalb der aktuellen Hierarchieebene an.

Aufgabe: Ausführen des Programms

a) Starten Sie die Simulation.

b) Führen Sie ihr Programm in der Konsole aus. Testen Sie alle Funktionen.

2.9 PID-Regler

Der Roboter soll um eine Geschwindigkeitsregelung erweitert werden. Hierzu wird zunächst ein PID-Regler (Klasse `PIDController`) in einem neuen Projekt implementiert. Objekte dieser Klasse sollen für alle Regelaufgaben in den folgenden Aufgaben verwendet werden können. Deshalb wird der Regler als PID-Regler konzipiert. Über die Zuweisung der Verstärkungsfaktoren K_p, K_i und K_d für Proportional-, Integrations- und Differentiationsanteil wird der jeweilige Regler entsprechend des speziellen Anwendungsfalls erzeugt.

2.9.1 Grundlagen zur Regelung

Einem dynamischen System, auch Strecke genannt, soll durch die Stellgröße u(t) ein gewünschtes Verhalten für die Ausgangsgröße y(t) aufgeprägt werden. Dies muss meisten gegen den Einfluss von Störungen z(t) erfolgen. Im Falle des Roboters bildet der Weg von der Schnittstelle über den Motorcontroller zu den Motoren die Strecke des Systems. Den Motoren soll über die Stellgröße (Servosignal) eine bestimmte Geschwindigkeit aufgeprägt werden.

Für eine Regelung wird die Ausgangsgröße y(t), im Regelkreis auch Regelgröße genannt, gemessen. Ausgehend von diesem Messwert $y_{mess}(t)$ und der Führungsgröße w(t) wird die Regelabweichung e(t) berechnet:

$$e(t) = w(t) - y_{mess}(t)$$

Anschließend wird die Stellgröße u(t) über Integration und Differentiation der Regelabweichung berechnet:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

Die Stellgröße wird danach der Regelstrecke zugeführt und so der Wunschverlauf der Ausgangsgröße y(t) erzeugt. Der Ablauf ist in Abbildung 26 für ein ideales Messglied ($y_{mess}(t) = y(t)$) dargestellt.

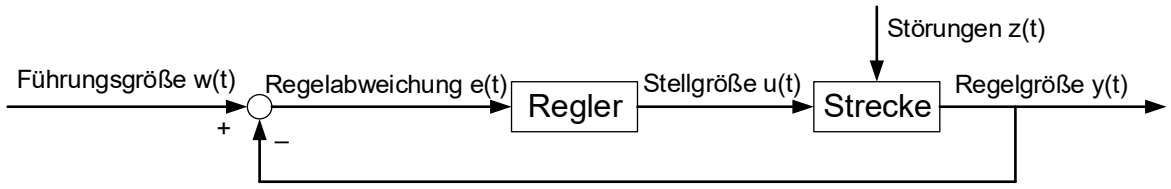


Abbildung 26: Aufbau eines Regelkreises

2.9.2 Realisierung am Digitalrechner

Die Regelung muss nun zeitdiskret für den Digitalrechner erfolgen. Die Berechnung der Regelabweichung eines jeden Zeitschritts k erfolgt wie im zeitkontinuierlichen Fall:

$$e_k = y_{soll} - y_{mess}$$

Die zeitdiskrete Integration wird durch das Summieren der Regelabweichungen über jeden Zeitschritt zu E_k und anschließender Multiplikation mit der Abtastzeit T_a [s] ausgeführt.

$$E_k = E_{k-1} + e_k$$

Zur zeitdiskreten Differentiation wird von der aktuellen Regelabweichung e_k die Regelabweichung des vorherigen Zeitschritts e_{k-1} subtrahiert und durch die Abtastzeit T_a geteilt. Alle Anteile der Stellgröße werden mit den Verstärkungsfaktoren verstärkt und aufaddiert:

$$u_k = K_p \cdot e_k + K_i \cdot T_a \cdot E_k + K_d \frac{e_k - e_{k-1}}{T_a}$$

In dieser Darstellung wird der aktuelle Zustand mit dem Index k und der Zustand bei einem vorherigen Zeitschritt mit k-1 gekennzeichnet.

2.9.3 Die Klasse PIDcontroller

Die Klasse `PIDcontroller` hat den in Abbildung 27 dargestellten Aufbau.

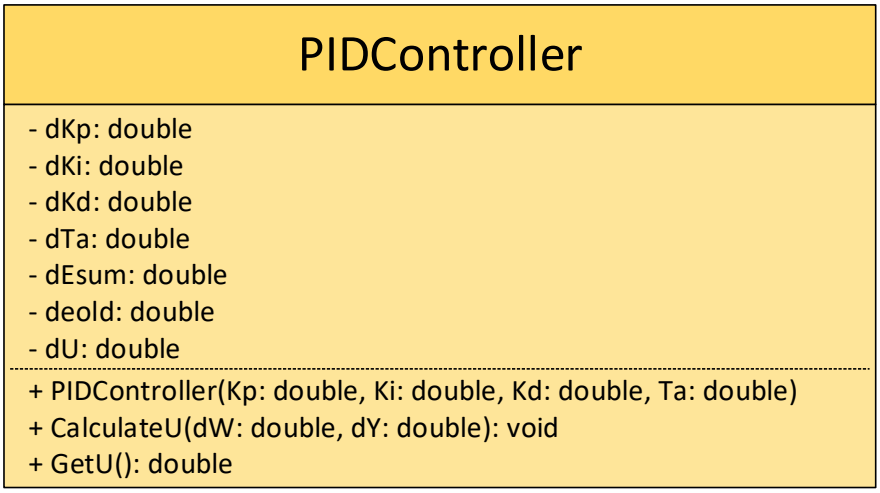


Abbildung 27: UML-Klassendiagramm der Klasse `PIDController`

Dem überladenen Konstruktor werden die Verstärkungsfaktoren sowie die Länge der Abtastzeitschritte [s] übergeben und in die entsprechenden Variablen gespeichert. Die Variablen für die Berechnung der Stellgröße (`dEsum`, `deold` und `dU`) werden mit 0 initialisiert. Die Methode `CalculateU()` berechnet die Stellgröße. Dabei werden die Führungsgröße (Soll-/Wunschwert) und die Regelgröße (Istwert) übergeben (siehe Abbildung 28).

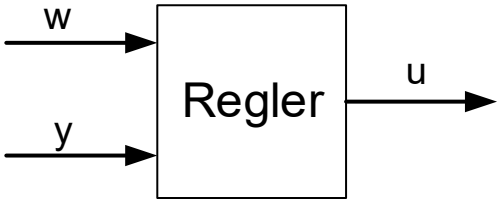


Abbildung 28: Umsetzung des Reglers in der Klasse `PIDcontroller`

Nach der Berechnung der Regelabweichung wird mit dem Regelalgorithmus die Stellgröße `dU` berechnet. Mit der Get-Methode `GetU()` wird die Stellgröße `dU` an das aufrufende Programm zurückgegeben.

Aufgabe: Regler

a) Erstellen Sie die Klasse `PIDController`, die beliebige Regler aus P- I-, und D-Reglern kombinieren kann. Der Klasse muss zur Erstellung eines beliebigen Reglers die Abtastzeit T_a zwischen den Regelgrößenmessungen und die Verstärkungsfaktoren K_P, K_i und K_d für Proportional-, Integrations- und Differentiationsanteil vorgegeben werden können.

b) Die Methode `calculateU()` soll aus dem Messwert y (y_{mess}) der Regelgröße und der Führungsgröße w (y_{soll}) die Stellgröße u generieren, aber keinen Rückgabewert besitzen (`void`). Der Regler soll jeweils nur eine Stellgröße berechnen, da später 2 Objekte des Reglers erzeugt werden. Die Stellgröße wird über eine get-Methode ausgelesen.

c) Zur Validierung des Reglers soll in der `main.cpp` ein Objekt des Reglers mit folgenden Parametern erstellt werden: K_p=1000, K_i=10, K_d=0.1, T_a=0.5. Anschließend sollen aus der Inputdatei `PIDControllerInput.txt`, die sich im Laufwerk `sefi` (`S:\`) befindet, Werte für die Soll- und Istwerte eingelesen werden (linke Spalte=Sollwerte, rechte Spalte=Istwerte). Aus diesen Werten soll jeweils die Stellgröße u in der Konsole ausgegeben und mit der Datei `PIDControllerVgl.txt` verglichen werden. Der Vergleich muss dabei nicht durch eine Methode durchgeführt werden. Ein Ingenieursblick reicht aus.

Hinweis: PIDControllerInput

Die Inputwerte sind jeweils mit einem Leerzeichen getrennt und als `double` gespeichert. Die Datei enthält insgesamt 10 Einträge.

2.10 Tastatursteuerung mit Regelung

Abschließend soll die Tastatursteuerung von vorhin mit dem Regler kombiniert werden. Dazu muss die Klasse `KeyboardControl` die Ansteuerung der Motoren mit Hilfe der `PIDController`-Klasse regeln.

Zunächst muss die Klasse `PIDController` (`PIDController.h` und `PIDController.cpp`) in das Projekt der Tastatursteuerung kopiert werden. Legen Sie dann mit Hilfe des überladenen Konstruktors zwei Objekte in der Klasse `KeyboardControl` an (eines für jeden Motor). Die Werte der Konstanten sind:

K_p=500.0, K_i=1850.0, K_d=0.0, T_a=0.04

Die Regler werden verwendet, um die Geschwindigkeiten der Motoren zu regeln. Als **Sollgeschwindigkeit** dient die benutzerspezifische Tastatureingabe. Die Geschwindigkeit der Räder wird gemessen und fließt als **Ist-Geschwindigkeit** ein. Die berechnete Stellgröße muss noch in den Nullwert des Servosignals verschoben werden. Dazu müssen jeweils **1500µs auf die Stellgröße u addiert** werden. Im letzten Schritt muss überprüft werden, ob die Signallängen des Servosignals den Bereich [1000, 2000] verlassen. Dazu kann eine `if`-Anweisung genutzt werden, die bei Verlassen des Bereichs die Signallänge auf den minimalen bzw. maximalen Wert setzt.

Aufgabe: Tastatursteuerung mit Regelung

a) Legen Sie zwei Regler für die Geschwindigkeit an und regeln Sie die Geschwindigkeiten der Motoren.

b) Testen Sie Ihr erweitertes Programm mit allen Funktionen.

Hinweis: Einfügen der Regler

Für jeden Motor soll ein eigener Regler verwendet werden. Fügen Sie dafür der Klasse `KeyboardControl` zwei Regler hinzu, die im Konstruktor mithilfe einer Initialisierungsliste initialisiert werden.

3 Tag 5

3.1 Einleitung

In dieser Aufgabe soll das automatische Abfahren von bestimmten Fahrmanövern implementiert werden. Dazu wird eine Ortsschätzung, die auf den gemessenen Geschwindigkeiten der Motoren basiert, integriert. Anschließend wird die Routine zum Abfahren eines Kreises und einer Acht realisiert. Hierfür werden vier Klassen benötigt, die in den folgenden Kapiteln implementiert werden:

- `PosEstimation`: Diese Klasse dient der Berechnung der Ist-Position des Roboters mit Hilfe der momentanen Geschwindigkeit und der vom vorherigen Durchgang gespeicherten Positionswerte.
- `Maneuver`: Dient sowohl der Berechnung der Koordinaten des Manövers, als auch der Berechnung der Soll-Geschwindigkeiten, um den nächsten Punkt der Koordinatenliste zu erreichen.
- `PIDController`: Diese Klasse stellt einen Regler zur Verfügung, der die Ist-Geschwindigkeit langsam auf die Soll-Geschwindigkeit angleicht.
- `RobotControl`: Diese Klasse verwaltet den Aufruf der Funktionen der einzelnen Klassen zur richtigen Zeit und ist ähnlich aufgebaut, wie die Klasse `KeyboardControl` des vorherigen Tages. Sie initialisiert Objekte der jeweiligen Klassen, besitzt eine Schnittstelle zur Kommunikation mit dem Benutzer (`Communicate()`) und stellt dem Interface eine Funktion zur iterativen Berechnung des berechneten Manövers (`Step()`) bereit.

Klassendiagramme zu allen Klassen, die im Folgenden beschrieben werden, sind im Kapitel 5.3 zu finden.

3.2 Positionsschätzung

Die Positionsschätzung berechnet (näherungsweise) die aktuelle Position (x-Koordinate und y-Koordinate) in einem kartesischen Koordinatensystem. Außerdem wird der aktuelle Winkel zur x-Achse in [rad] berechnet. Das Koordinatensystem wird am Startpunkt des Roboters initialisiert (siehe Abbildung 29)). Nach der Initialisierung des Ursprungs bleibt dieser fixiert. Der Roboter bewegt sich innerhalb des aufgespannten Koordinatensystems.

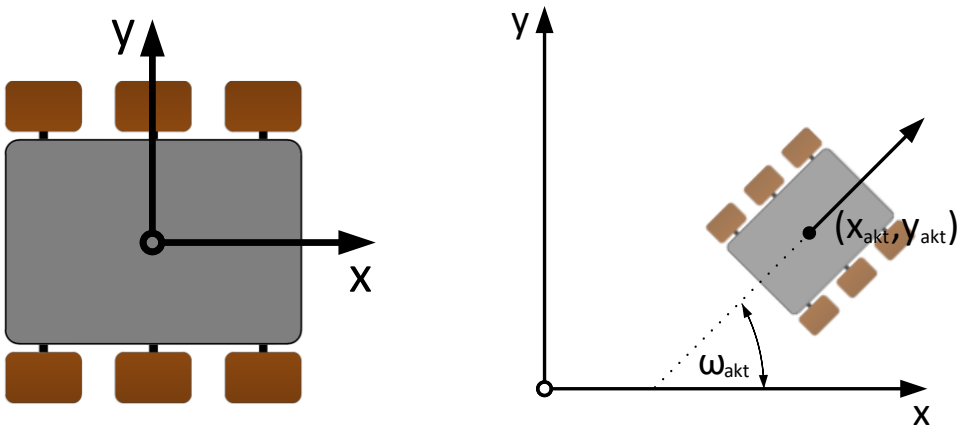


Abbildung 29: Ausrichtung des Koordinatensystems bei Initialisierung und Roboter bei der Fahrt

3.2.1 Die Klasse PosEstimation

Die Klasse `PosEstimation` hat den in Abbildung 30 dargestellten Aufbau.

Mit Hilfe der Methode `Reset()` wird der Koordinatenursprung zurückgesetzt. Die Methode `PredictPosition()` berechnet die aktuelle Position und die Raumrichtung nach dem in Kapitel 3.2.2 erläuterten Prinzip. `GetPosition()` liefert die Adresse des Ergebnisarrays (x[3]) zurück, sodass die Werte in der Aufrufenden Methode verwendet werden können.

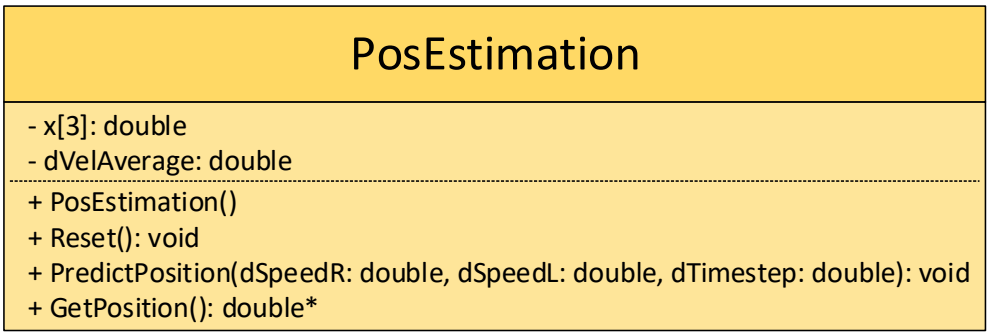


Abbildung 30: UML-Klassendiagramm der Klasse `PosEstimation`

3.2.2 Die Funktion PredictPosition(…)

Die Positionsschätzung erfolgt auf Basis der gemessenen Geschwindigkeitswerte der beiden Motoren. Es wird davon ausgegangen, dass die Motoren des Roboters in jedem Zeitintervall T (Einheit [s]) konstant mit der Geschwindigkeit drehen, die zu Beginn des Intervalls gemessen wurde. Zur Berechnung der x-, y-Koordinaten sowie der mittleren Geschwindigkeit v und der Raumrichtung w wird das implizite Euler-Verfahren (auch Rückwärts-Euler-Verfahren) verwendet. Im Vergleich zur Position beim vorherigem Zeitschritt (x_{k-1}, y_{k-1}) hat sich der Roboter einen Zeitschritt lang mit der mittleren Geschwindigkeit v_{k-1} bewegt, die im vorherigen Zeitschritt berechnet wurde. Die Strecke s, um die der Roboter sich bewegt hat, ist somit:

$$s = v_{k-1} \cdot T$$

Diese Strecke wird nun entsprechend der Achsenrichtungen mit dem Sinus bzw. Kosinus der Raumrichtung des letzten Zeitschritts w_{k-1} gewichtet, sodass folgende Formeln für die aktuelle Position (x_k, y_k) resultieren:

$$x_k = x_{k-1} + v_{k-1} \cdot T \cdot \cos w_{k-1}$$
$$y_k = y_{k-1} + v_{k-1} \cdot T \cdot \sin w_{k-1}$$

Die Winkelgeschwindigkeit des Roboters lässt sich aus der Differenz der Geschwindigkeiten der beiden Motoren und dem Abstand der Räder wie folgt berechnen:

$$\omega = \frac{v_{rechts} - v_{links}}{Abstand\ der\ Räder}$$

Der Abstand der Reifen beträgt bei dem Roboter 0,3m, sodass für die Richtung w_k, in die sich der Roboter bewegt, resultiert:

$$w_k = w_{k-1} + T \cdot \left(\frac{v_{rechts} - v_{links}}{0{,}23}\right)$$

Damit der Winkel w_k nicht beliebig große Werte annimmt, wird er auf den Einheitskreis]-π, π] begrenzt. Dazu kann der Modulo-Befehl `double fmod (double a, double b)` aus der Bibliothek <`cmath`> verwendet werden. Dieser gibt den Rest der Division a/b zurück. Im Gegensatz zum einfachen %-Befehl werden auch Kommawerte betrachtet.

z.B.: `fmod`(5.3, 4) == 1.3.

Da der Wert π innerhalb des Intervalls liegen soll, kann der Modulo-Befehl nicht mit b=π durchgeführt werden, sondern es bedarf eines kleinen Workarounds (bitte beide Punkte beachten und umsetzen):

1. Der Winkel w_k wird über den Modulo-Befehl auf das Intervall]-2π, 2π[eingeschränkt:

$$w_k = fmod(w_k, 2 \cdot \pi)$$

2. Für alle Werte w_k, die größer als π sind, werden 2π abgezogen. Für jeden Wert der kleiner oder gleich –π ist, werden 2π addiert. Dieser Schritt lässt sich mit einer einfachen `if`-Anweisung durchführen.

Die mittlere Geschwindigkeit (`dVelAverage`) im aktuellen Zeitschritt berechnet sich folgendermaßen:

$$v_k = \frac{v_{rechts} + v_{links}}{2}$$

Hinweis: Reihenfolge der Berechnung

Bitte die Reihenfolge beachten, wie sie hier dargestellt wird!

Aufgabe: Positionsschätzung

Erstellen Sie die Klasse `PosEstimation`. Nutzen Sie in der Klasse die Standardbibliothek <`cmath`>.

a) Sehen Sie eine `void`-Methode vor, welche aus der Geschwindigkeit des rechten und linken Motors (v_{rechts}, v_{links}) den aktuellen Ort im Koordinatensystem und die Raumrichtung berechnet. Nutzen Sie zum Speichern der Ergebnisse ein `double` Array mit drei Elementen (`x[0]` für die x-Koordinate, `x[1]` für die y-Koordinate und `x[2]` für die Raumrichtung). `vlinks`, `vrechts` und die Zeitschrittlänge `T` werden als `double`-Werte übergeben.

b) Die Rückgabe der berechneten Werte (x-/y-Koordinate und Raumrichtung) erfolgt über die separate `Get`-Methode.

c) Implementieren Sie außerdem die `Reset`-Methode, in der die Attribute der Klasse auf den Null-Zustand zurückgesetzt werden.

d) Die Validierung der Positionsschätzung erfolgt ähnlich der des Reglers. Erstellen Sie eine main.cpp, die ein Objekt der Positionsschätzung enthält. Lesen Sie anschließend die Werte aus der Inputdatei `PosEstimationInput.txt` ein (erste Spalte = `vrechts`, zweite Spalte = `vlinks`, dritte Spalte = Zeitschritt). Die Werte der Inputdatei sind jeweils durch ein Leerzeichen getrennt. Mit diesen Werten soll nun eine Positionsschätzung durchgeführt werden, Die Ergebnisse der Schätzung werden in der Konsole ausgegeben und mit dem Werten der Vergleichsdatei `PosEstimationVgl.txt` verglichen (linke Spalte x-Koordinate, mittlere Spalte y-Koordinate, dritte Spalte Raumrichtung).

Beachten Sie die Hinweise.

Hinweis: Berechnung der Position

Achten Sie darauf, dass die Zeitschrittlänge T, mit dem Zeitintervall übereinstimmt, in dem die `step`-Methode zyklisch aufgerufen wird. Die Funktionsweise dieser Methode wird in Kapitel 3.4.2 beschrieben.

Die Funktionen zur Berechnung von sin- und cos-Werten ist wie folgt definiert:

```
double sin (double x);

double cos (double x);
```

In der Bibliothek <`cmath`> ist die Zahl π hinterlegt und kann mit `M_PI` in Berechnungen verwendet werden.

3.3 Autonomes Manöver fahren

In diesem Aufgabenteil soll eine Klasse entwickelt werden, die es dem Roboter ermöglicht ein vorprogrammiertes Manöver (im Praktikum ein Kreis und eine Acht) zu fahren. Dazu werden x- und y-Koordinaten sowie die Geschwindigkeit, mit der die Koordinaten angefahren werden sollen, in einer Liste gespeichert. Der Ursprung des

Koordinatensystems wird über die Klasse `PosEstimation` festgelegt. Mit den drei Informationen ist es möglich, beliebige Koordinatenpaare anzufahren und die Geschwindigkeit für jeden neuen Weg individuell anzupassen. Mit der Methode `LogList(std::string sDatei)` können die berechneten Punkte in eine Log-Datei geschrieben werden. Mit der Log-Datei kann die Funktionsweise der Koordinatenberechnung überprüft werden.

Anhand der Daten der Liste und der Positionsschätzung berechnet der Roboter, mit welcher Geschwindigkeit die beiden Motoren sich drehen müssen, damit die Punkte mit der gewünschten Geschwindigkeit angefahren werden. Die Berechnung wird in der Methode `CalcManeuverSpeed()` durchgeführt. Auf Basis der aktuellen Position und der Position, zu der gefahren werden soll, werden Geschwindigkeitsanteile für eine Rotation und eine Translation des Roboters berechnet. Aus diesen beiden Anteilen wird schließlich die Wunschgeschwindigkeit für die beiden Motoren zusammengesetzt. Die Methode `GetManeuverSpeed()` wird erst später verwendet.

Abbildung 31 zeigt den Aufbau der Klasse `Maneuver`:

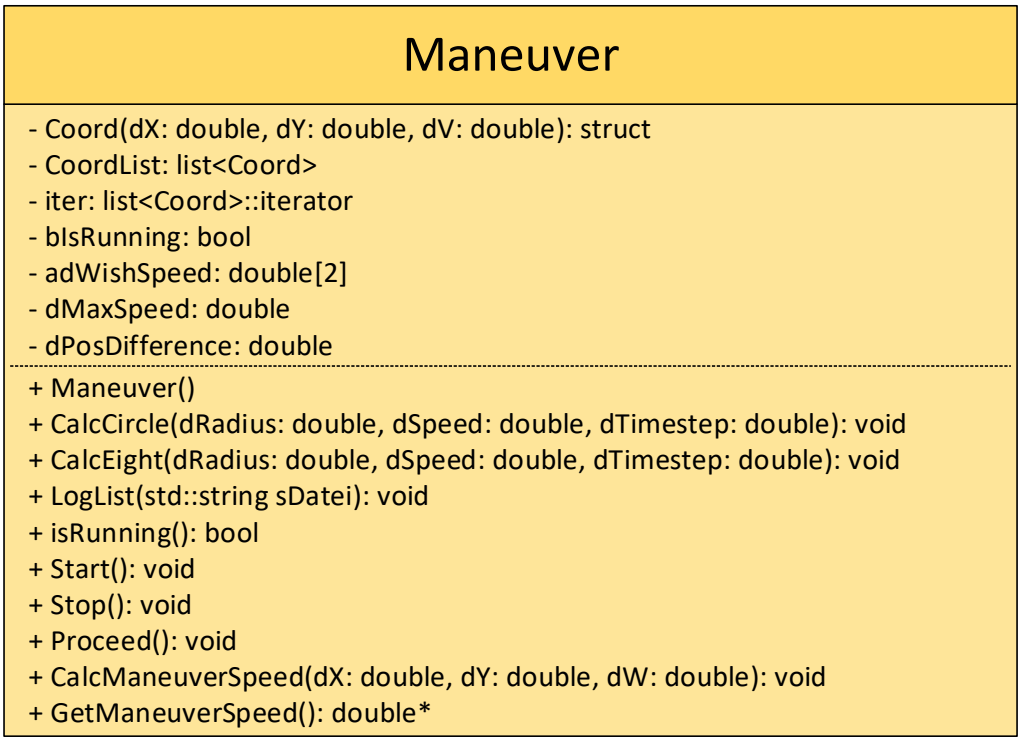


Abbildung 31: UML-Klassendiagramm der Klasse `Maneuver`

3.3.1 Koordinatenliste erzeugen

Um ein Manöver fahren zu können, muss eine Liste mit folgenden Einträgen generiert werden:

- x-Koordinate
- y-Koordinate
- Geschwindigkeit

Mit diesen drei Informationen ist es möglich beliebige Koordinatenpaare anzufahren und die Geschwindigkeit für jeden neuen Weg individuell anzupassen. Damit der

Roboter das Manöver sauber ausführt, soll für jeden Zeitschritt ein neues Punktepaar berechnet werden. Für den Kreis und die Acht wird der Geschwindigkeitseintrag konstant gehalten, damit der Kreis bzw. die Acht gleichmäßig durchfahren wird.

Um die Manöverinformationen in einer Liste zu speichern, wird eine Struktur benötigt. Diese muss Einträge für die x- und y-Koordinate sowie die Geschwindigkeit enthalten (siehe Abbildung 31). Aus dieser Struktur wird anschließend eine Liste generiert, welche durch die entsprechenden Methoden mit Werten gefüllt wird.

Die Formeln zur Berechnung der Koordinaten für die Manöver Kreis und Acht lauten wie folgt:

Kreis:

$$x = R \cdot \sin(counter \cdot \frac{v}{R} \cdot T)$$
$$y = R \cdot (1 - \cos(counter \cdot \frac{v}{R} \cdot T)$$

Acht:

$$\left.\begin{aligned} x &= R \cdot \sin(counter \cdot \frac{v}{R} \cdot T) \\ y &= R \cdot (1 - \cos(counter \cdot \frac{v}{R} \cdot T) \end{aligned}\right\} \text{1. Teilkreis}$$

$$\left.\begin{aligned} x &= R \cdot \sin(counter \cdot \frac{v}{R} \cdot T) \\ y &= (-R) \cdot (1 - \cos(counter \cdot \frac{v}{R} \cdot T) \end{aligned}\right\} \text{2. Teilkreis}$$

Da zu jedem Zeitschritt ein neuer Punkt (x-Koordinate, y-Koordinate und Geschwindigkeit) benötigt wird, muss die Zeitbasis T, mit der die `Step`-Methode arbeitet, in [s] übergeben werden. In den oben angegebenen Formeln sind die Geschwindigkeit v in [m/s] und der Radius R in [m] angegeben.

3.3.2 Koordinatenliste ausgeben

Die gefüllte Koordinatenliste soll zur Kontrolle in eine Log-Datei geschrieben werden. Hierfür wird die Methode `LogList` erstellt. Jedes Koordinatenpaar der Liste wird in eine Zeile der Log-Datei geschrieben. Dabei werden x- und y-Koordinate mit einem Tabstopp getrennt (`\t`).

Aufgabe: Koordinatenliste

Beachten Sie den Hinweis!

a) Erstellen Sie die Klasse `Maneuver` und implementieren Sie jeweils eine Methode zum Erzeugen eines Kreises und einer Acht. Der Radius R, die gewünschte Geschwindigkeit v sowie die Zeitbasis T werden an die jeweilige Methode übergeben. Benutzen Sie für die Berechnungen der Koordinaten die Standardbibliothek <`cmath`>. Denken Sie daran, die Liste vor jedem neuen Füllen zu löschen.

b) Programmieren Sie die Methode `LogList(std::string sDatei)`. Diese schreibt die Koordinatenliste in eine Textdatei. Testen Sie anschließend die Funktionen zur Erstellung eines Kreises und einer Acht in einer main-Funktion. Erzeugen Sie dazu einen Kreis mit den Parametern R=2.0, v=0.3, T=0.04 und Loggen die Werte in eine Log-Datei. Benennen Sie diese anschließend in `LogFileCircle.txt`. Erstellen Sie auf die gleiche Weise die Datei `LogFileEight.txt`, die die Daten einer Acht (R=5.0, v=1.0, T=10.0) enthält.

Prüfen Sie zunächst die Logdateien auf Plausibilität. Importieren Sie anschließend das Projekt Kapitel5_Koordinatenliste_Test und kopieren Sie die von ihnen erstellten Dateien in dieses Projekt. Durch Ausführen des Kontrollprojektes können Sie überprüfen, ob die Berechnung des Kreises bzw. der Acht korrekt ist.

Hinweis: for-Schleife zur Berechnung der (Teil-)Kreise

Benutzen Sie zum Berechnen des jeweiligen (Teil-)Kreises die folgende for-Schleife:

```
for(int counter = 1; counter<(int)((2*M_PI)/((v/R)*T)); counter++)
```

Beachten Sie bitte die korrekte Syntax.

3.3.3 Das Flag bIsRunning

Das Flag `bIsRunning` wird für die Steuerung der Manöverfahrt benötigt. Anhand des Flags lässt sich zu jedem Zeitschritt bestimmen, ob ein Manöver gefahren werden soll oder nicht. Ist das Flag `true` werden die Geschwindigkeiten für die beiden Motoren bestimmt. Bei `false` werden die Motoren angehalten und auf weitere Befehle gewartet. Um es in der Klasse `RobotControl` abrufen zu können, wird mit einer entsprechenden get-Methode `isRunning()` gearbeitet.

3.3.4 Die Methoden Start, Stop, Proceed

Die Methode `Start()` soll ein Manöver von Beginn der Koordinatenliste starten. Deshalb wird in `Start()` der Iterator auf den Anfang der Liste initialisiert und außerdem das Flag `bIsRunning` auf `true` gesetzt. Beim Aufrufen von `Stop()` wird das Flag auf `false` gesetzt und das Manöver somit unterbrochen. In `Proceed` wird hingegen nur das Flag auf `true` gesetzt, sodass ein unterbrochenes Manöver wiederaufgenommen wird.

Aufgabe: Kontrollmethoden

a) Implementieren Sie das Flag `bIsRunning` und die entsprechende Übergabemethode.

b) Implementieren Sie die Methoden `Start`, `Stop` und `Proceed` entsprechend der beschriebenen Struktur in die Klasse `Maneuver`.

3.3.5 Geschwindigkeitsberechnung während der Manöverfahrt

Für das Abfahren der Koordinatenliste muss zu jedem Zeitschritt die notwendige Geschwindigkeit für den rechten und linken Motor berechnet werden. Zur Berechnung der Geschwindigkeiten müssen folgende Schritte durchlaufen werden (siehe auch Abbildung 32):

1. Positionsvergleich:
   Zu Beginn muss überprüft werden, ob der Roboter die Sollposition `xsoll` und `ysoll` im letzten Zeitschritt mit hinreichender Genauigkeit erreicht hat. Als maximale Abweichung (`dPosDifference`) sollten 2 cm gewählt werden. Ist die Sollposition hinreichend genau erreicht, wird der Iterator der Koordinatenliste ein Listenelement weitergeschaltet. Ansonsten wird das Listenelement beibehalten und im nächsten Zeitschritt erneut angefahren.

Hinweis: Einheiten

Die Abweichung muss in Metern mit den x- und y-Koordinaten verglichen werden, da diese ebenfalls in Metern berechnet werden.

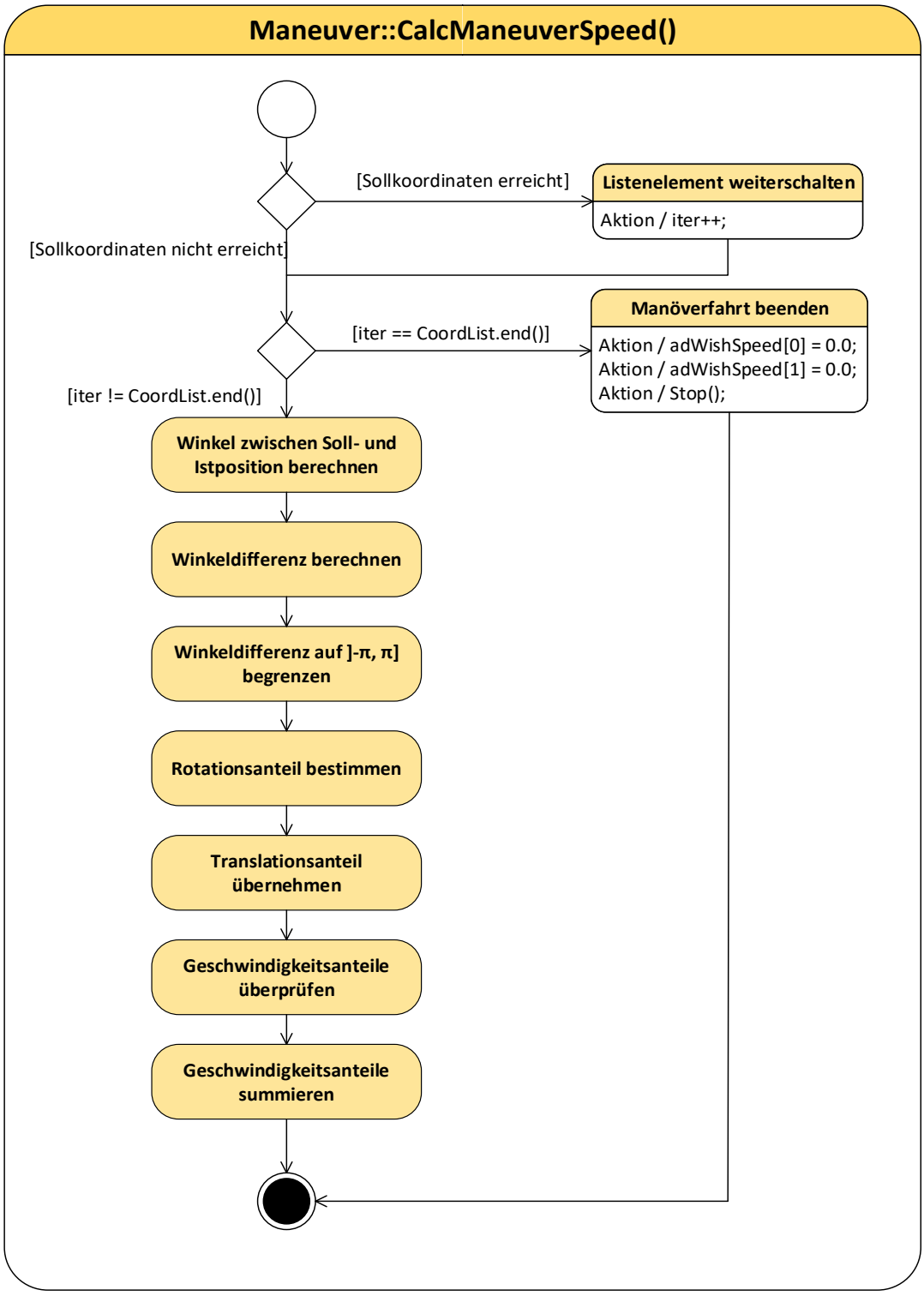


Abbildung 32: UML-Zustandsdiagramm der Funktion zum Manöverfahren

2. Überprüfen, ob das Ende der Liste erreicht wurde:
   Sobald das Ende der Liste erreicht ist, werden die Wunschgeschwindigkeiten auf null gesetzt und das Manöver gestoppt (Aufruf von `Stop()`). Ist das Ende der Liste noch nicht erreicht, werden die Schritte 3 bis 0 ausgeführt.

3. Winkel zwischen Soll- und Ist-Position berechnen:

Für den Winkel φ zwischen der aktuellen Position (x_{akt}, y_{akt}) und anzufahrenden Position (x_{soll}, y_{soll}) gilt folgender Zusammenhang (siehe hierzu auch Abbildung 33):

$$\varphi = \tan^{-1}\left(\frac{y_{soll} - y_{akt}}{x_{soll} - x_{akt}}\right)$$

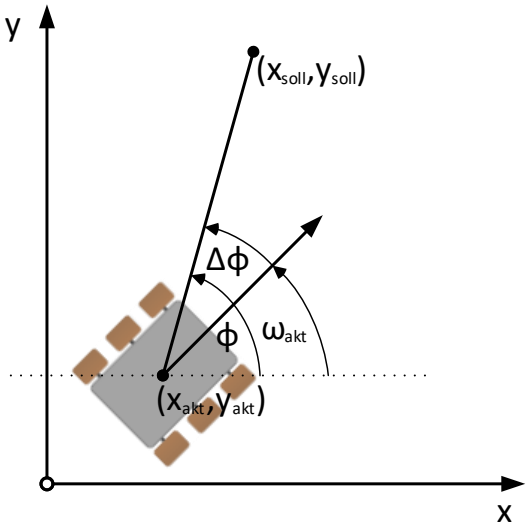


Abbildung 33: Winkelberechnung

Die Koordinaten `xsoll` und `ysoll` werden aus der Liste entnommen. Die Werte für `xakt` und `yakt` sind Übergabeparameter der Methode `CalcManeuverSpeed()`.

Hinweis: Berechnung der Geschwindigkeit

Nutzen Sie die Funktion **`double atan2(double y, double x)`** aus <`cmath`>, welche Werte im Bereich [-π, π] zurückgibt. Die Funktion `atan2(y, x)` entspricht der Argument-Funktion aus der Rechnung mit komplexen Zahlen.

Bitte achten Sie darauf, dass die Funktionsdefinition zuerst den **y-** und dann den **x-Wert** erwartet und dass der aktuelle Positionswert von dem Sollwert abgezogen wird.

4. Winkeldifferenz berechnen:
   Die Differenz Δφ zwischen der aktuellen Raumrichtung des Roboters w_{akt} und der notwendigen Raumrichtung φ gibt an wie stark sich der Roboter drehen muss, um geradlinig auf die Soll-Position zuzufahren. Die verschiedenen Winkel und deren Zusammenhang können Sie Abbildung 33 entnehmen. Für den Winkel Δφ gilt nachfolgende Berechnungsvorschrift:
   $$\Delta\varphi = \varphi - w$$
5. Winkeldifferenz auf]-π, π] begrenzen:
   Bei der Berechnung von Δφ können Winkel auftreten für die gilt | Δφ | > π. In diesen Fällen ist es sinnvoller in entgegengesetzter Richtung zu drehen. Auf diese Weise wird sichergestellt, dass der Roboter maximal um 180° rotiert.

Zur Begrenzung muss 2π auf alle Winkel, für die Δφ ≤ -π gilt, addiert und 2π von allen Winkeln, für die gilt Δφ > π, subtrahiert werden.

6. Rotationsanteil bestimmen:
   Aus der Winkeldifferenz Δφ wird der Rotationsanteil der Geschwindigkeit berechnet. Dazu wird die Winkeldifferenz mit dem Proportionalitätsfaktor 2 [m/(s rad)] gewichtet. Anschließend soll über eine `if`-Anweisung sichergestellt werden, dass der Rotationsanteil den Bereich von [-0.5,0.5] m/s nicht unter- bzw. überschreitet. (Achten Sie hierbei bitte auf die Vorzeichen!)

7. Translationsanteil übernehmen:
   Als Translationsanteil der Geschwindigkeit wird der Wert aus der Liste übernommen.

8. Geschwindigkeitsanteile überprüfen:
   Damit der Roboter die vorgegebenen Punkte erreicht, muss sichergestellt werden, dass der Rotationsanteil in jedem Fall in die Geschwindigkeiten eingeht. Der Translationsanteil kann also notfalls unterdrückt werden. Der Ablauf der Überprüfung ist in Abbildung 34 dargestellt. Achten Sie bitte auf die korrekte Nutzung der Vorzeichen und Operatoren (+ & -).

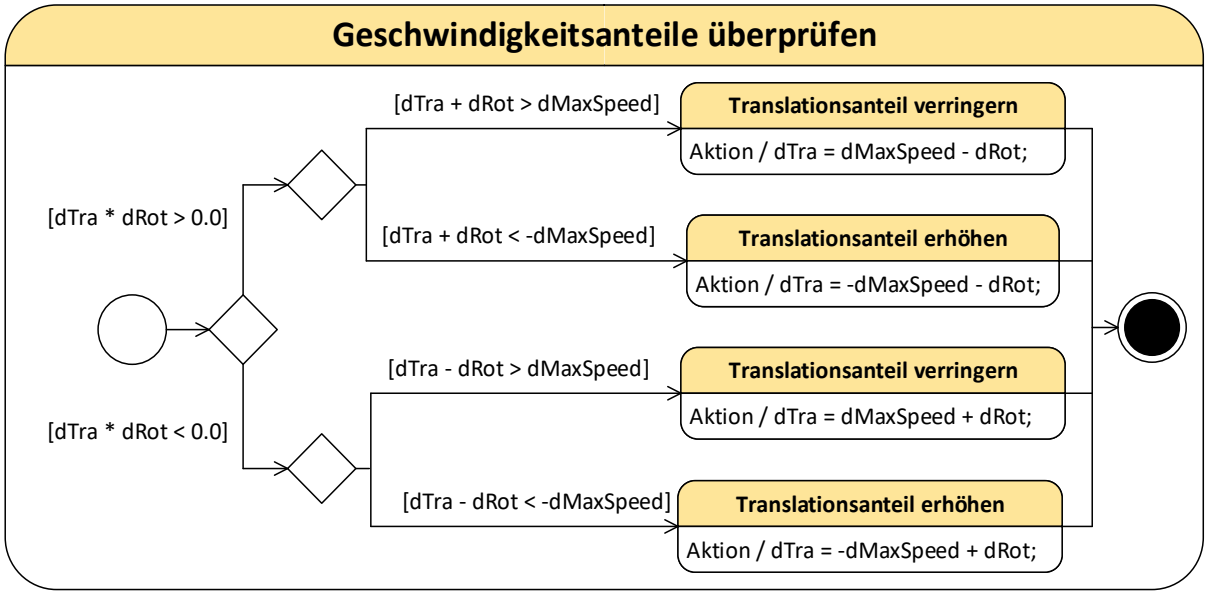


Abbildung 34: UML-Zustandsdiagramm der Geschwindigkeitsanteilsüberprüfung

9. Geschwindigkeiten summieren:

```
adWishSpeed[0] = dTra + dRot;
adWishSpeed[1] = dTra - dRot;
```

Hinweis: adWishSpeed

Für die Wunschgeschwindigkeit werden die Rotations- und Translationsanteile summiert. Dabei soll ein positiver Rotationsanteil eine Drehung in mathematisch positiver Richtung (Gegen-Uhrzeiger-Sinn) erzeugen. Er muss also auf den Translationsanteil des rechten Motors addiert und vom Translationsanteil des linken Motors subtrahiert werden.

Achten Sie darauf, dass die Wunschgeschwindigkeiten den Maximalbetrag von 0,5 m/s nicht übersteigen.

Diese Aufgabe ist sehr fehleranfällig! Vergewissern Sie sich, dass Sie die korrekten Vorzeichen und Variablen verwenden und letztere **sinnvoll initialisiert** sind.

Aufgabe: Geschwindigkeitsberechnung und -übergabe

a) Erweitern Sie die Klasse `Maneuver` um den Algorithmus zur Berechnung der Wunschgeschwindigkeiten. Die Methode `CalcManeuverSpeed()` muss die Berechnung der Geschwindigkeiten für die beiden Motoren gewährleisten. Ihr wird die aktuelle Lage im Koordinatensystem (x_{akt}, y_{akt} und w_{akt}) aus der Ortschätzung übergeben. Zum Speichern der berechneten Geschwindigkeiten dient ein `double`-Array mit zwei Elementen (Geschwindigkeit rechts, Geschwindigkeit links).

b) Integrieren Sie außerdem die Methode `GetManeuverSpeed()`, die einen Pointer auf die Adresse des Geschwindigkeitsarrays übergibt. Mit dieser Funktion können die berechneten Geschwindigkeiten in der Klasse `RobotControl` ausgelesen werden.

3.4 Integration des autonomen Manövers in die Robotersteuerung

Abschließend werden die zuvor programmierten Klassen in die Robotersteuerung integriert. Hierzu wird die Klasse `RobotControl` erstellt, welche die verschiedenen Funktionalitäten zusammenführt. Das UML-Klassendiagramm in Abbildung 35 zeigt den groben Aufbau der Klasse `RobotControl`.

Die Klasse `RobotControl` enthält Objekte vom Typ `InterfaceSIM`, `Maneuver`, `PosEstimation` und `PIDcontroller`. Außerdem werden der `transferPointer` und die `transferFunction()`, die bereits für die Tastatursteuerung benutzt wurden, benötigt. Das Flag `isActive` und die dazugehörige `get`-Methode `isActive()` dienen dazu, in der `main.cpp` zu überprüfen, ob die Manöverfahrt genutzt wird oder das Programm beendet werden soll.

Den Kern der Robotersteuerung bilden die Funktionen `Step()` und `Communicate()`. Wie die Namen schon andeuten, ist `Communicate()` für die Interaktion mit dem Benutzer verantwortlich. `Step()` hingegen liest Sensorsignale aus, verarbeitet diese und sendet neue Signale an die Aktoren. Die beiden Funktionen werden im Folgenden detailliert beschrieben.

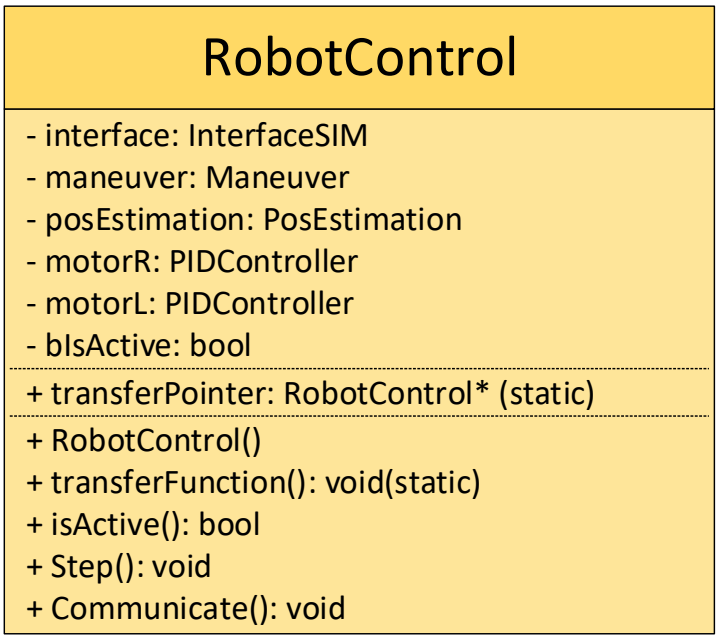


Abbildung 35: UML-Klassendiagramm der Klasse `RobotControl`

Hinweis: Klassendiagramm RobotControl

Das UML-Diagramm Abbildung 35 enthält nur die essentielle Funktionalität der Klasse `RobotControl`. Es kann vorteilhaft sein, für die Speicherung/Verarbeitung von Funktionswerten zusätzliche Variablen zu verwenden.

3.4.1 Die Methode *Communicate*

Für die Kommunikation wird, wie auch bei der Tastatursteuerung, die Bibliothek `ncurses` verwendet. Das Vorgehen für die Verwendung verläuft analog zur Tastatursteuerung.

Die Methode `Communicate` führt einen Dialog mit dem Benutzer und liest dabei Radius, Geschwindigkeit sowie das gewünschte Manöver ein. Entsprechend der Wahl des Benutzers wird ein Kreis oder eine Acht erzeugt. Für diesen Teil der Kommunikation wird die Bibliothek <`iostream`> verwendet, da mit dieser auch Werte des Typs `double` eingelesen werden können.

Anschließend wird die Ortsschätzung initialisiert (alle Werte auf 0 gesetzt), die Bibliothek `ncurses` gestartet und die `Step`-Methode eingeschaltet (siehe Kapitel 2.7). Ab diesem Zeitpunkt wird zum Einlesen des Benutzerwunsches nur noch die Funktion `getch()` aus `curses.h` verwendet. Der Benutzer erhält die Möglichkeit, in einer Schleife periodisch das Manöver zu starten, zu stoppen oder fortzusetzen. Bei jeder Aktion soll die entsprechende Methode aus `Maneuver` aufgerufen werden. Die Schleife wird solange ausgeführt, bis „abbrechen“ (‘q‘) gedrückt wird. Dann wird das Manöver gestoppt und die Schleife wird verlassen.

In der nachfolgenden Schleife wird überprüft, ob der Roboter auch wirklich angehalten hat (vgl. Tastatursteuerung). Danach wird die Methode `Step()` abgeschaltet und die Bibliothek `ncurses` mit `endwin()` beendet.

Abbildung 36 zeigt die Struktur von `Communicate()`.

In der `main.cpp` wird die Methode `Communicate()` in einer `do-while`-Schleife solange ausgeführt, bis das Flag `bIsActive` auf `false` gesetzt wird. Dies geschieht, sobald der Benutzer bei der Frage, ob ein Manöver gefahren werden soll, ablehnt.

Aufgabe: Die Methode Communicate

Implementieren Sie die Methode `Communicate`. Orientieren Sie sich an Abbildung 36 und der gegebenen Beschreibung. Als `timestep` verwenden Sie hierfür wieder 0.04. Schreiben Sie außerdem die main.cpp.

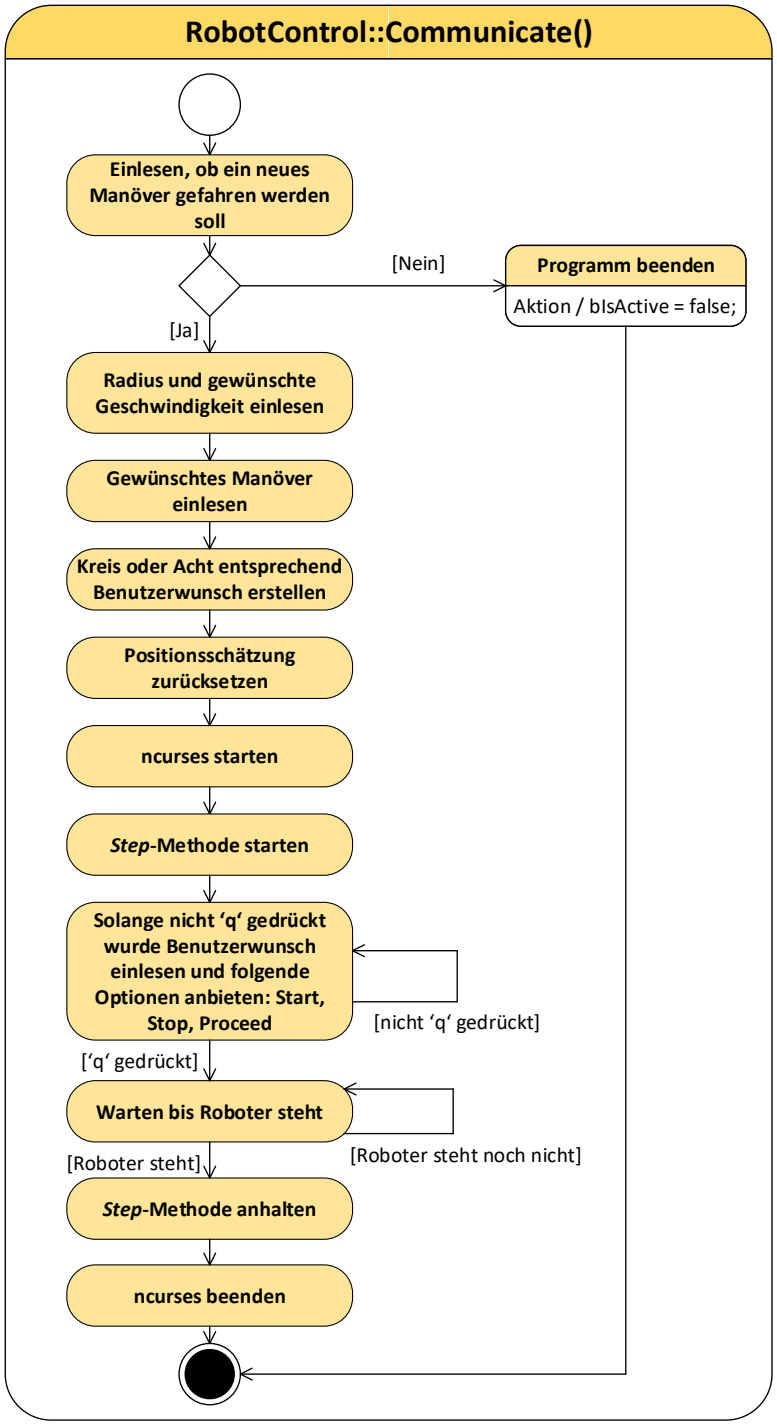


Abbildung 36: UML-Zustandsdiagramm der Methode `Communicate`

3.4.2 Die Methode Step

Die `Step`-Methode hat folgenden Aufbau:

Zunächst werden die aktuellen Geschwindigkeiten des Roboters abgerufen und zwischengespeichert.

Ist die Manöversteuerung aktiv (`bIsRunning = true`) werden diese Geschwindigkeiten an die Ortsschätzung übergeben, die den momentanen Ort und Winkel des Roboters bestimmt. `Maneuver` verwendet den aktuellen Ort und Winkel, um die neuen Wunschgeschwindigkeiten des Roboters zu berechnen. Diese Geschwindigkeiten werden über **zwei** Regler (einen für jeden Motor) geregelt und in Signallängen konvertiert. Es folgt die Überprüfung der Zulässigkeit der Signallängen. Hierbei muss sichergestellt werden, dass die Signallängen im Intervall [1000µs, 2000µs] liegen.

Ist die Manöversteuerung nicht aktiv, werden die Signallängen auf den Referenzwert (1500 µs) gesetzt.

Abschließend werden die Signallängen an den Roboter übermittelt.

Abbildung 37 zeigt das Schema der Methode `Step`.

Aufgabe: Die Methode Step

Implementieren Sie die Methode *Step*. Orientieren Sie sich an Abbildung 37 und der gegebenen Beschreibung.

Kompilieren Sie anschließend ihr Programm und führen Sie es in der Simulation aus.

Beachten Sie die Hinweise.

Hinweis: Reglerparameter

Für die Regler der beiden Motoren sollten folgende Reglerparameter gewählt werden:

Kp = 500.0, Ki = 100.0, Kd = 0.0;

Hinweis: Aufrufen der Step()-Methode

„Step-Methode starten“ bezeichnet keinen Aufruf von Step(), sondern das Einfügen der in Kapitel 2.7 gezeigten Codezeilen „sigprocmask(…)“.

Hinweis: Parallelisierung

Bitte denken Sie an daran, die im Kapitel 2.7 beschriebenen Schritte zur Parallelisierung ebenfalls für die Klasse RobotControl durchzuführen.

Hinweis: Geeignete Testwerte

Als geeignete Testwerte haben sich folgende Werte bewährt:

Radius = 0.3, Geschwindigkeit = 0.3;

Diese gelten ebenfalls für die spätere Hardwareausführung im Kapitel 4.3. Andere Werte können ebenfalls gewählt werden, es sollte bei der Parameterwahl aber darauf geachtet werden, dass der Radius in Metern und die Geschwindigkeit in m/s angegeben wird und 0.5 die maximale Geschwindigkeit der Roboter ist.

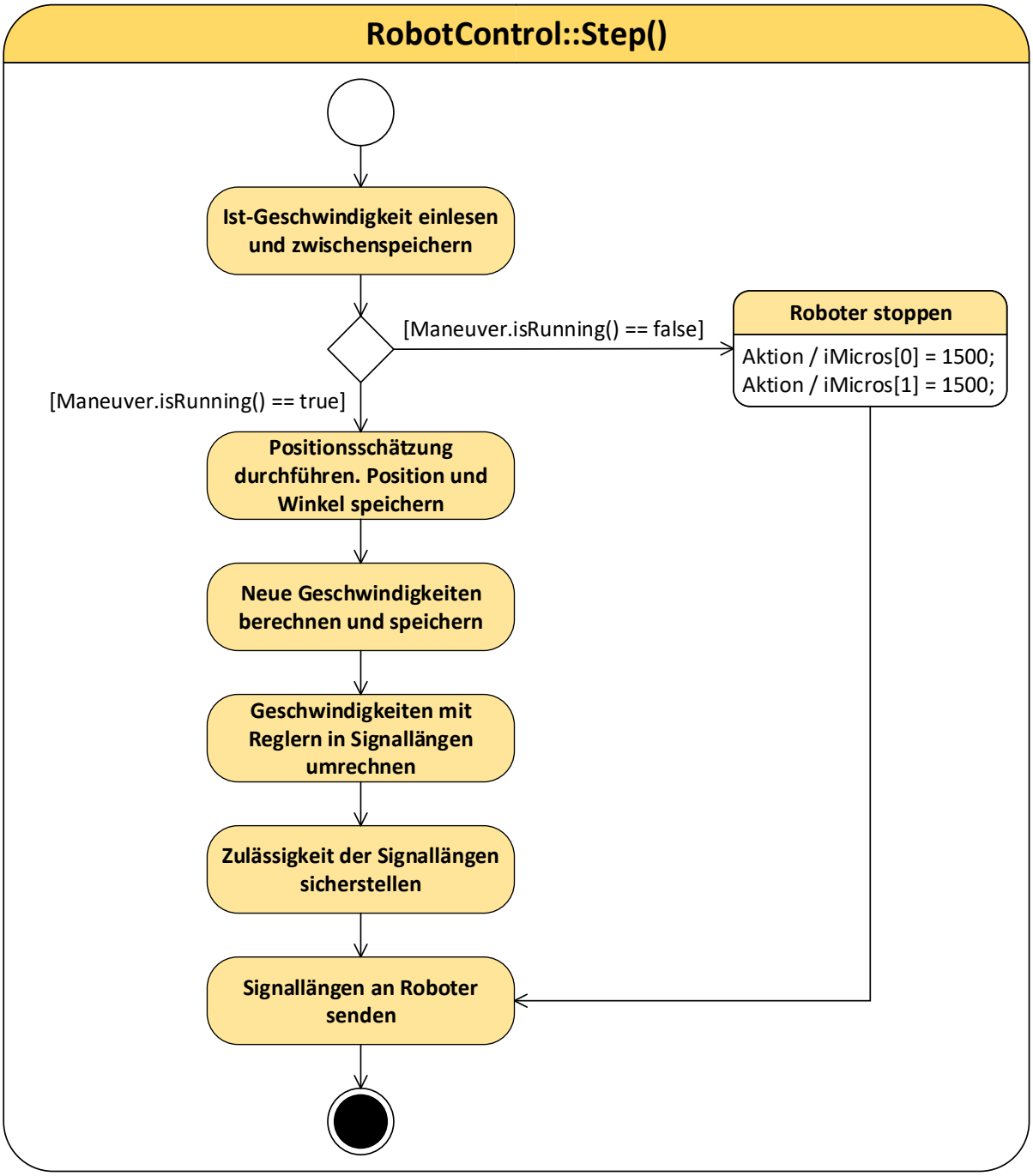


Abbildung 37: UML-Zustandsdiagramm der Methode *Step*

4 Tag 6

4.1 Cross-Compiling-Projekt für den Roboter

Die in diesem Praktikum benutzte IDE Eclipse läuft in einer x86-Umgebung. Der Raspberry Pi basiert dagegen auf der ARMv6 Architektur. Um dennoch mit einem x86 Rechner Software für den Raspberry Pi entwickeln zu können, muss Eclipse also in der Lage sein, Projekte für die ARM Architektur kompilieren zu können. Dies geschieht mithilfe einer sogenannten *Cross Compiling Toolchain*. Diese gibt es für den Raspberry Pi bereits fertig zusammengestellt zum Herunterladen. Auf der virtuellen Maschine ist die *Toolchain* bereits installiert und kann sofort benutzt werden.

4.1.1 Ein Cross-Compiling-Projekt erstellen

Folgende Schritte sind notwendig, um ein neues Projekt für den Raspberry Pi zu erstellen:

1. *File* → *New* → *C++* wählen.
2. Ein neues Fenster öffnet sich, in dem man zunächst einen Projektnamen eingeben muss. Als Projekttyp ist *Executable* → *Empty Project* und als Toolchain *Cross GCC* auszuwählen.
3. Danach zweimal auf *Next* klicken.
4. Im *Cross GCC Command* Fenster müssen folgende Eintragungen gemacht werden:
5. im *Cross Compiler prefix* Feld

   ```
   arm-linux-gnueabihf-
   ```

6. im *Cross Compiler path* Feld

   ```
   /home/pi/rpi/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/
   ```

7. Mit *Finish* wird die Konfiguration beendet.

Im Projektexplorer kann nun eine neue `.cpp` erstellt werden. Zum jetzigen Zeitpunkt könnte man bereits einen Code für den Raspberry Pi kompilieren. Damit die Roboter-Hardware über den Raspberry Pi angesteuert werden kann, werden allerdings noch externe Bibliotheken und eine Verbindung zum Senden der Binary-Datei benötigt. Die Einstellungen dafür werden in den nächsten Absätzen beschrieben.

4.1.2 Einbinden von externen Bibliotheken

Für den Roboter werden externe Bibliotheken benötigt, um die GPIO-Pins ansteuern oder eine Tastatureingabe ohne Enter-Taste einlesen zu können. Weil nicht für eine x86-Umgebung sondern den Raspberry Pi kompiliert wird, müssen die Bibliotheken für die ARMv6 Architektur benutzt werden. Das bedeutet, dass die Bibliotheken zuerst für die Zielarchitektur kompiliert werden müssen. Dies kann entweder direkt auf dem Raspberry Pi oder mit einer Cross Compiling Toolchain geschehen. Für das Praktikum wurden die Bibliotheken bereits fertig für die ARMv6 Architektur kompiliert und liegen auf der virtuellen Maschine im Ordner `/home/pi/RaspberryLibs`.

Wie bereits in Kapitel 22 des Grundlagenskripts erklärt wurde, müssen dem Compiler und Linker die Pfade zu den Dateien der externen Bibliotheken angegeben werden:

1. Der Pfad der Headerdateien lautet:

   ```
   /home/pi/RaspberryLibs/include
   ```

2. und ist in das *include*-Verzeichnis unter dem Reiter *GNU C++* (linke Seite) einzutragen (siehe Abbildung 38).
3. In der Kartei *Libraries* müssen folgende Bibliotheken eingebunden werden:

   ```
   rt
   pthread
   pigpio
   ncurses
   ```

4. Der *Library Path* lautet:

   ```
   /home/pi/RaspberryLibs/lib
   ```

Nun erkennt Eclipse die Syntax für die neue Bibliothek und der Code kann nach dem kompilieren auf die GPIO Pins zugreifen.

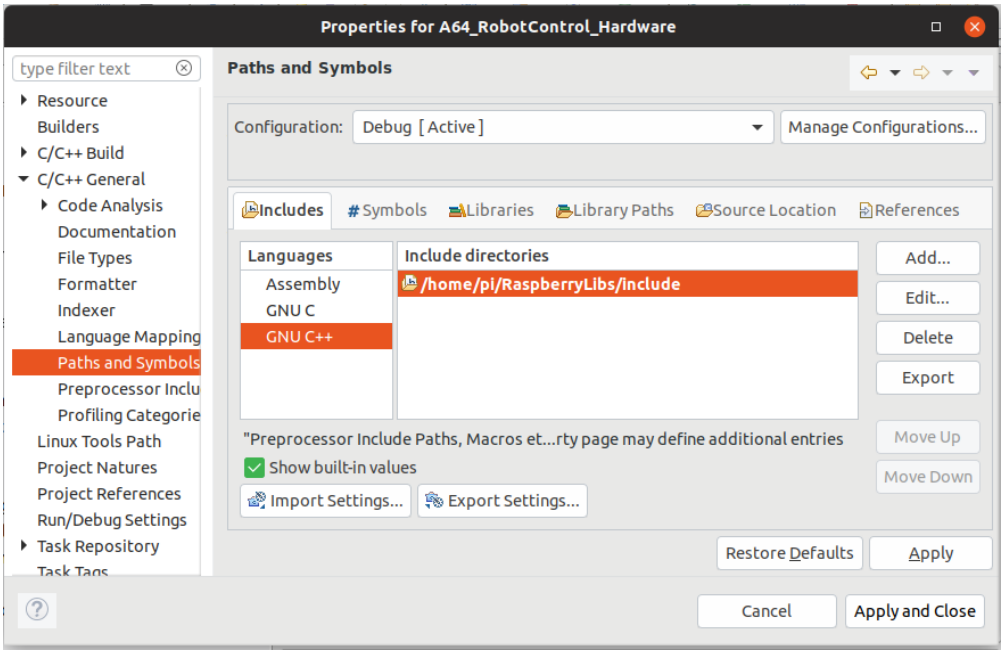


Abbildung 38: Include directories angeben

4.2 Beispielprojekt: Cross-Compiling-Projekt

Zunächst soll in einem ersten Cross-Compiling-Projekt direkt mit den GPIO-Pins interagiert werden. Dafür werden Sie die PIGPIO-Bibliothek verwenden. Nachfolgend ist der Programmablauf beschrieben.

4.2.1 PIGPIO-Bibliothek für GPIO lesen und schreiben

Benötigte Funktionen aus der PIGPIO-Bibliothek sind:

- Initialisieren der Bibliothek vor dem ersten Aufruf
  `gpioInitialise();`
- Ansteuern des Motorcontrollers mit Servo-Signal
  `gpioServo(int PIN, int iSignalLaenge);`
- Anhalten der PIGPIO-Bibliothek
  `gpioTerminate();`

4.2.1.1 Einzubindende Header-Dateien

Folgende Header sind in das Programm einzubinden. Achten Sie auf die veränderte Syntax bei ncurses.

```
#include <pigpio.h>
#include <ncurses/curses.h>
#include <unistd.h>
```

4.2.1.2 Ablauf des Programms in der `main()`-Funktion

- Initialisieren Sie zu Beginn der `main`-Funktion die PIGPIO-Bibliothek.
- Initialisieren Sie für rechts und links je eine `int` Variable mit dem Mittelwert des Servosignals (ca. 1500 [µs]).
- Initialisieren Sie eine `char` Variable, in welcher später die Tastatureingabe zwischengespeichert wird.
- Starten Sie die Bibliothek `ncurses`, wie sie es bereits in den vorherigen Blättern gelernt haben
- Starten Sie eine `while`-Schleife, die wie in den bisherigen Aufgaben bis zu der Eingabe eines Zeichens zum Beenden der Schleife durchgeführt wird, z.B. ‘q‘.
- Zu Beginn der `while`-Schleife wird mit `getch()` aus `<curses.h>` ein Zeichen in die zuvor initialisierte Variable eingelesen.
- Die Auswertung des Zeichens soll nur erfolgen, wenn wirklich ein Zeichen eingelesen wurde.
- Prüfen Sie mit einer `if`-Anweisung, ob ein bestimmtes Zeichen zum Beenden des Programms eingelesen wurde z.B. ‘q‘.
- Ansonsten prüfen Sie zum Beispiel mittels einer `switch-case`-Anweisung, ob das Zeichen für vorwärts, rückwärts, drehen nach links, drehen nach rechts, break oder beenden eingelesen wurde. Erhöhen, verringern oder setzen Sie die Servosignal-Variablen entsprechend. z.B.:
  - Vorwärts: `iRechts += 10;` und `iLinks += 10;`
  - Drehen nach rechts: `iRechts -= 10;` und `iLinks += 10;`

- Stellen Sie anschließend sicher, dass die Signallängen, die an die Roboter gesendet werden, den gültigen Bereich von [1000,2000] nicht unter- oder überschreiten (siehe 2.6 Die Step-Funktion).
- Sofern das Zeichen für Stopp (‘q‘) oder Beenden (‘b‘) gedrückt wurde, soll der Initial-/Mittelwert des Servosignals eingestellt werden.
- Nach der Veränderung der Variablen soll mit `gpioServo` die entsprechend oben eingestellten Werte an die Servo-Funktion übergeben werden, sodass der Befehl von den Motoren umgesetzt wird (z.B.: `gpioServo(17, iLinks)`:
  - PIN 17 linker Motor
  - PIN 18 rechter Motor
- Nach der `while`-Schleife muss sichergestellt werden, dass die letzten Signallängen noch an die Motoren weitergegeben werden können. Da dies eine gewisse Zeit in Anspruch nimmt, muss der Ablauf der main-Funktion kurzzeitig gestoppt werden. Eine einfache Pause von einer Sekunde lässt sich mit dem Befehl `sleep(1)` bewerkstelligen. Zur Nutzung dieser Funktion muss die Headerdatei `unistd.h`, wie oben angegeben, eingebunden werden.
- Anschließend muss die Bibliothek `ncurses` beendet werden.
- Am Ende der `main()`-Funktion muss zum regulären Beenden `gpioTerminate()` aufgerufen werden.

Aufgabe: Cross-Compiling Projekt

Erstellen Sie ein Projekt nach Kapitel Beispielprojekt: Cross-Compiling-Projekt und fügen Sie eine `.cpp`-Datei hinzu, welche die main-Funktion beinhalten soll. Die main-Funktion enthält die Steuerung, die den ebenfalls oben beschrieben Ablauf hat.

Hinweis: Cross-Compiling-Aufgabe

Folgend finden Sie eine Anleitung zur Inbetriebnahme Ihres Codes auf dem realen Roboter. Diese benötigen Sie zum Ausführen Ihres Projektes aus Aufgabe 4.2.

Beachten Sie den Hinweis nach dem folgenden Kapitel.

4.2.2 Kopieren und Ausführen des kompilierten Codes auf den Raspberry Pi

Um den Code auf den Raspberry Pi zu übertragen, wird allerdings noch eine Verbindung mit diesem benötigt. Die WLAN-Antenne des Roboters spannt ein WLAN-Netz auf. Wenn man sich mit diesem verbindet, kann anschließend eine SSH-Verbindung zum Raspberry Pi aufgebaut werden.

Hierfür werden die schon bekannten Ubuntu Anwendungen *Nautilus* (Ubuntu Datei Explorer) und das *Linux Terminal* benötigt. Alternativ dazu können auch Windows Anwendungen (z.B. *WinSCP* und *PuTTY*) verwendet werden. Diese Alternative wird an dieser Stelle nicht weiter behandelt, kann aber jederzeit bei Bedarf durch die Tutoren erklärt werden.

Das Vorgehen zur Kommunikation mit dem Roboter wird in den folgenden Kapiteln beschrieben. Vorher müssen folgende Schritte umgesetzt werden:

1. Verbinden Sie Ihren Windows mit dem WLAN des Roboters. Hierfür erhalten Sie von den Tutoren einen WLAN-Stick.
2. Das Passwort zum Netzwerk der Roboter erhalten Sie ebenfalls von den Tutoren.
3. Nach erfolgreicher Verbindung muss in den *Virtual Machine Settings* des *VMware Players* der *Network Adapter* auf *NAT* umgeschaltet werden. Dies sollte bereits vorab eingestellt sein. Ist dies nicht der Fall, kann dies unter dem Pfad *Player* → *Manage* → *Virtual Machine Settings…* → *Hardware* → *Network Adapter* umgestellt werden.
4. Folgen Sie den Beschreibungen der folgenden zwei Kapitel.

4.2.2.1 Kopieren der Dateien auf den Raspberry Pi

Um die kompilierten Dateien auf den Raspberry Pi zu kopieren verwenden wir den Ubuntu Explorer Nautilus. Um Dateien vom Ubuntu System auf das System des Roboters zu übertragen, muss zunächst eine Verbindung mit diesem hergestellt werden:

1. Öffnen Sie den Nautilus Explorer.
2. Klicken Sie in der Ordner Wahl auf den Link + *Andere Orte* (Siehe Abbildung 39).

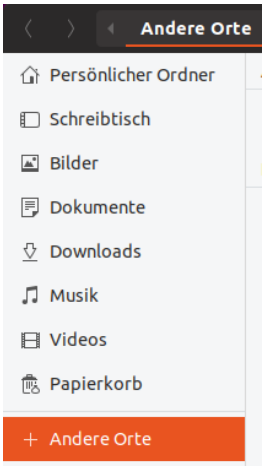


Abbildung 39 Link + Andere Orte
im Nautilus Explorer

3. Geben Sie im Feld *Mit Server Verbinden* (siehe Abbildung 40) folgendes ein, um eine Explorer Verbindung mit dem Roboter herzustellen:

```
ssh://root@192.168.1.20
```

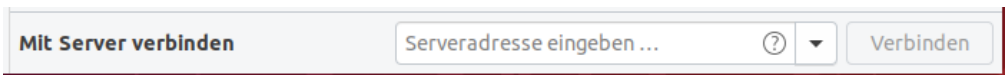


Abbildung 40 Textfeld Mit Server verbinden

4. Das Passwort für den Roboter lautet:

   raspberry

5. Sobald Sie sich verbunden haben, können Sie die Binary Dateien aus dem Debug Ordner der jeweiligen Projekte mittels Drag-and-Drop in das *root*-Verzeichnis des Zielsystems kopieren.

4.2.2.2 Ausführen der Binary-Datei

Im Folgenden wird beschrieben, wie man ein ausführbares Programm auf dem Raspberry Pi startet. Hierfür benötigten Sie das Linux Terminal, um auf den Raspberry Pi zugreifen zu können. Folgende Schritte sind dafür notwendig:

1. Öffnen Sie das Linux Terminal.
2. Geben Sie den Befehl `ssh root@192.168.1.20` ein. Dieser Befehl startet die SSH Verbindung zum Ziel mit der IP-Adresse 192.168.1.20 und meldet sich dort als Benutzer root an.
3. Das Passwort für den Roboter lautet:

   raspberry

4. Besteht nun eine Verbindung mit dem Roboter, können Sie genauso wie bei ncurses erklärt, Programme starten und bedienen. Die Eingabe des Befehls

   ```
   ./Dateiname
   ```

   in das Terminal startet das Programm mit dem Dateinamen `Dateiname`.

Hinweis: Ausführen des Programms auf Roboter

Bevor Sie den Roboter fahren lassen, ist es unbedingt notwendig, dass alle Grundfunktionen funktionieren, insbesondere `break`! Um dies zu testen, können Sie den Roboter vorsichtig hochheben und das Programm ausführen. Greifen Sie den Roboter niemals am Deckel oder Überrollbügel, sondern ausschließlich an der Bodenwanne. Es sollte während des Betriebs immer eine Person in der Nähe des Roboters sein, um diesen im Notfall hochheben zu können.

Hinweis: Gültige Signallängen

Achten Sie darauf, dass die Signallängen, die an den Roboter gesendet werden, den Bereich [1000,2000] nicht verlassen.

4.3 Anpassen der Simulationsprogramme für die Hardware

Nachdem Sie nun eine erste Steuerung für den Roboter geschrieben und getestet haben, sollen zum Abschluss des Praktikums die in den vorangegangenen Kapiteln erstellten Steuerungen für die Simulation so angepasst werden, dass sie auf dem Roboter lauffähig sind. Dazu wird die Simulationsschnittstelle durch eine Hardwareschnittstelle ersetzt. Außerdem müssen einige Änderungen im Programmcode vorgenommen werden, da die Parallelisierung bei den beiden Schnittstellen unterschiedlich verläuft.

4.3.1 Die Hardwareschnittstelle

Die Schnittstelle zur Hardware ist in Form der Klassen `InterfaceHW` gegeben. Die nötigen `.h`- und `.cpp`-Dateien heißen:

`InterfaceHW.h`

`InterfaceHW.cpp`

`InterfaceHWArdupi.h`

`InterfaceHWArdupi.cpp`

`InterfaceHWHall.h`

`InterfaceHWHall.cpp`

`InterfaceHWSpeed.h`

`InterfaceHWSpeed.cpp`

und befinden sich im dem `Netzlaufwerk` (`S:\`). Um die Schnittstelle zu benutzen, werden diese Dateien in das entsprechende Projekt kopiert und `InterfaceHW.h` in die jeweilige `Control`-Klasse eingebunden. Einem Objekt vom Typ `InterfaceHW` stehen folgende Methoden zur Verfügung:

1. `double* GetInput()` und `void SetOutputs(int* iMicros)`

   die genauso verwendet werden können, wie die entsprechenden Methoden der Simulationsschnittstelle (`InterfaceSIM`).

   Hinweis: Aufruf von GetInput()

   Die Methode `GetInput()` darf nicht außerhalb der definierten Zeitschritte der `Step()`-Methode aufgerufen werden, da die Zeitbasis in die Berechnung der aktuellen Geschwindigkeit eingeht und es somit zu Fehlern kommen würde.

2. `void Initialize(double Zeitschrittlänge)`

   zur Initialisierung des Hardwareschnittstellenobjektes. Als Übergabeparameter wird die Länge der Zeitschritte in [s], in denen die Methode `Step()` aufgerufen wird, benötigt. Die Zeitschrittlänge dient dazu die mittels der Inkrementalgeber gezählten Flanken in eine Geschwindigkeit umzurechnen. Achten Sie darauf, dass die Zeitbasis bei allen genutzten Funktionen gleich ist. Diese Methode muss im Konstruktor der jeweiligen `Control`-Klasse aufgerufen werden.

3. `void SetTimerfunction(int timer, double Zeitbasis, void *Funktionspointer)`

   zur Erstellung einer timerbasierten Funktion, die zu diskreten Zeitschritten ausgeführt wird. Es können 10 Timer verwendet werden. Dazu wird dem Übergabeparameter timer ein Wert zwischen 0 und 9 übergeben. Die Zeitbasis in [s] gibt an in welchen Zeitschritten die Funktion aufgerufen werden soll. Dem Funktionspointer muss die Adresse einer `void`-Funktion übergeben werden, die zu den diskreten Zeitschritten ausgeführt werden soll.

4.3.2 Erzeugung der parallelen Step()-Methode

Das Parallelisieren der Funktionen `Step()` und `Communicate()` erfolgt mit der Methode `SetTimerfunction()` der Hardwareschnittstelle. Die Methode sollte am Ende des Konstruktors (nach der Initialisierung des Hardwareschnittstellenobjektes) der jeweiligen `Control`-Klasse aufgerufen werden, so dass danach die Methoden `Communicate()` und `Step()` parallel ausgeführt werden. Einzig die Zuweisung des Transferpointers auf die Klasse selbst (`transferPointer = this`) muss noch danach erfolgen. Die Benutzung der Methode `SetTimerfunction()` sollte wie folgt aussehen:

```
<NameDesSchnittstellenObjekts>.SetTimerfunction(0, 0.04,
(&transferFunction));
```

Die parallele Ausführung der `Step()`-Methode durch `SetTimerfunction()` muss nicht gestartet oder beendet werden. Die folgenden Zeilen in der jeweiligen `Control`-Klasse werden also **nicht** mehr benötigt:

```
1  sigprocmask(SIG_UNBLOCK, &<NameDesSchnittstellenObjekts>.mask, nullptr);
2  sigprocmask(SIG_BLOCK, &<NameDesSchnittstellenObjekts>.mask, nullptr);
```

Hinweis: Reglerparameter

Die Regelungsparameter des PID Reglers müssen angepasst werden. Bitte Stellen Sie folgende Parameter ein:

$K_p = 500$, $K_i = 1850$, $K_d = 0$;

Aufgabe: Tastatursteuerung (Hardware)

Erstellen Sie ein neues Projekt mit den entsprechenden Einstellungen für das Cross-Compiling und kopieren Sie anschließend die Source-Files Ihrer geregelten Tastatursteuerung in das Projekt. Tauschen Sie die Simulationsschnittstelle gegen die Hardwareschnittstelle. Nehmen Sie danach die oben beschriebenen Änderungen im Code vor und führen Sie das Programm auf der Hardware aus.

Aufgabe: Manöversteuerung (Hardware)

Erstellen Sie ein neues Projekt mit den entsprechenden Einstellungen für das Cross-Compiling (Nehmen Sie nicht das Projekt der vorherigen Aufgabe!). Gehen Sie anschließend wie in der vorherigen Aufgabe vor und wandeln so Ihre Manöversteuerung für die Hardware ab. Testen Sie die Steuerung abschließend auf der Hardware.

Alternativaufgabe: Kombinierte Tastatur- und Manöversteuerung (Hardware)

Achtung: Diese Aufgabe kann alternativ zu den vorhergehenden zwei bearbeitet werden.

Erstellen Sie ein neues Projekt mit den entsprechenden Einstellungen für das Cross-Compiling. Kopieren Sie die benötigten .cpp und .h-Dateien aus den vorhergehenden Aufgaben in dieses Projekt. Achten Sie darauf, Doppelungen zu vermeiden (z.B. keine zwei Controller.cpp-Dateien).

Schreiben Sie nun eine Main-Funktion, in der mit einer Tastatur-Eingabe zwischen beiden Steuerungsarten gewählt werden kann. Dabei soll das entsprechende Steuerungsobjekt erst nach der Auswahl erstellt werden (Gleichzeitig existierende Objekte von KeyboardControl und RobotControl führen zu Problemen mit dem Interface).

Das Praktikum „Industrielle Softwareentwicklung für Ingenieur:innen / C++“ ist nun zu Ende. Wir hoffen, dass es Ihnen Spaß gemacht hat. Über Anmerkungen und Kritik freuen wir uns sehr. Zögern Sie also nicht, uns anzusprechen.

5 Anhang

5.1 ASCII-Tabelle

ASCII steht für „**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange“ und dient zur Kodierung von Zeichensätzen. Tabelle 11 listet die nicht druckbaren Steuerzeichen wie Zeilenvorschub, Tabulator oder Protokollzeichen auf. Tabelle 12 und Tabelle 13 listen die druckbaren Zeichen wie das Alphabet in Groß- und Kleinschreibung, die arabischen Ziffern und Satzzeichen auf.

Dez	Hex	ASCII	Anmerkung	Dez	Hex	ASCII	Anmerkung
0	0x00	*NUL*	Null	17	0x11	*DC1*	Device Control 1
1	0x01	*SOH*	Start of Heading	18	0x12	*DC2*	Device Control 2
2	0x02	*STX*	Start of Text	19	0x13	*DC3*	Device Control 3
3	0x03	*ETX*	End of Text	20	0x14	*DC4*	Device Control 4
4	0x04	*EOT*	End of Transmission	21	0x15	*NAK*	Negative Acknowledge
5	0x05	*ENQ*	Enquiry	22	0x16	*SYN*	Synchronous Idle
6	0x06	*ACK*	Acknowledge	23	0x17	*ETB*	End of Transmission Block
7	0x07	*BEL*	Bell	24	0x18	*CAN*	Cancel
8	0x08	*BS*	Backspace	25	0x19	*EM*	End of Medium
9	0x09	*HT*	Horizontal Tab	26	0x1A	*SUB*	Substitute
10	0x0A	*LF*	Line Feed	27	0x1B	*ESC*	Escape
11	0x0B	*VT*	Vertical Tab	28	0x1C	*FS*	File Separator
12	0x0C	*FF*	Form Feed	29	0x1D	*GS*	Group Separator
13	0x0D	*CR*	Carriage Return	30	0x1E	*RS*	Record Separator
14	0x0E	*SO*	Shift Out	31	0x1F	*US*	Unit Separator
15	0x0F	*SI*	Shift In	…	…	…	…
16	0x10	*DLE*	Data Link Escape	127	0x7F	*DEL*	Delete

Tabelle 11: ASCII-Tabelle (Steuerzeichen)

Dez	Hex	ASCII	Dez	Hex	ASCII	Dez	Hex	ASCII
32	0x20		55	0x37	7	78	0x4E	N
33	0x21	!	56	0x38	8	79	0x4F	O
34	0x22	“	57	0x39	9	80	0x50	P
35	0x23	#	58	0x3A	:	81	0x51	Q
36	0x24	$	59	0x3B	;	82	0x52	R
37	0x25	%	60	0x3C	<	83	0x53	S
38	0x26	&	61	0x3D	=	84	0x54	T
39	0x27	‘	62	0x3E	>	85	0x55	U
40	0x28	(	63	0x3F	?	86	0x56	V
41	0x29	)	64	0x40	@	87	0x57	W
42	0x2A	*	65	0x41	A	88	0x58	X
43	0x2B	+	66	0x42	B	89	0x59	Y
44	0x2C	,	67	0x43	C	90	0x5A	Z
45	0x2D	-	68	0x44	D	91	0x5B	[
46	0x2E	.	69	0x45	E	92	0x5C	\
47	0x2F	/	70	0x46	F	93	0x5D	]
48	0x30	0	71	0x47	G	94	0x5E	^
49	0x31	1	72	0x48	H	95	0x5F	_
50	0x32	2	73	0x49	I	96	0x60	`
51	0x33	3	74	0x4A	J	97	0x61	a
52	0x34	4	75	0x4B	K	98	0x62	b
53	0x35	5	76	0x4C	L	99	0x63	c
54	0x36	6	77	0x4D	M	100	0x64	d

Tabelle 12: ASCII-Tabelle (druckbare Zeichen)

Dez	Hex	ASCII	Dez	Hex	ASCII

101	0x65	e	114	0x72	r
102	0x66	f	115	0x73	s
103	0x67	g	116	0x74	t
104	0x68	h	117	0x75	u
105	0x69	i	118	0x76	v
106	0x6A	j	119	0x77	w
107	0x6B	k	120	0x78	x
108	0x6C	l	121	0x79	y
109	0x6D	m	122	0x7A	z
110	0x6E	n	123	0x7B	{
111	0x6F	o	124	0x7C	|
112	0x70	p	125	0x7D	}
113	0x71	q	126	0x7E	~

Tabelle 13: ASCII-Tabelle (druckbare Zeichen)

5.2 Verwendung von Namenskonventionen bei der Programmierung

Die Verwendung von Namenskonventionen verbessert die Lesbarkeit des Codes und unterstützt beispielsweise das Auffinden von Fehlern sowie die Änder- und Erweiterbarkeit des Codes. Die Einhaltung von Namenskonventionen in Entwicklerteams tragen wesentlich zur Verständlichkeit und somit zu Qualität und Fehlerreduzierung in Softwareprojekten bei und sind somit besonders im industriellen Einsatz unerlässlich.

Beispiele für Namenskonventionen sind die 'Ungarische Notation' oder die 'Java-Code-Convention'. Über diesen hinaus gilt, dass in Codes und in Kommentaren die folgenden Buchstaben **nicht** verwendet werden: **ä**, **ö**, **ü**, **ß**.

Für das Praktikum wird eine Namenskonvention, angelehnt an die „Ungarische Notation“, eingesetzt.

5.2.1 Aufbau

Die Namenskonvention setzt sich zusammen aus Präfix und Bezeichner. Das Präfix wird klein geschrieben. Bei der Benennung des Bezeichners ist die Pascalschreibweise (Erster Buchstabe jedes verketteten Wortes wird großgeschrieben) zu verwenden. Bsp.: szBlackColor

5.2.2 Präfixe für Datentyp

1. Die Verwendung von Präfixen soll den Datentyp der Variable verdeutlichen.
2. Die Präfixe können auch kombiniert werden. Die Bezeichnung *paszTabelle* bezeichnet beispielsweise einen Zeiger auf ein Array null-terminierter Strings.

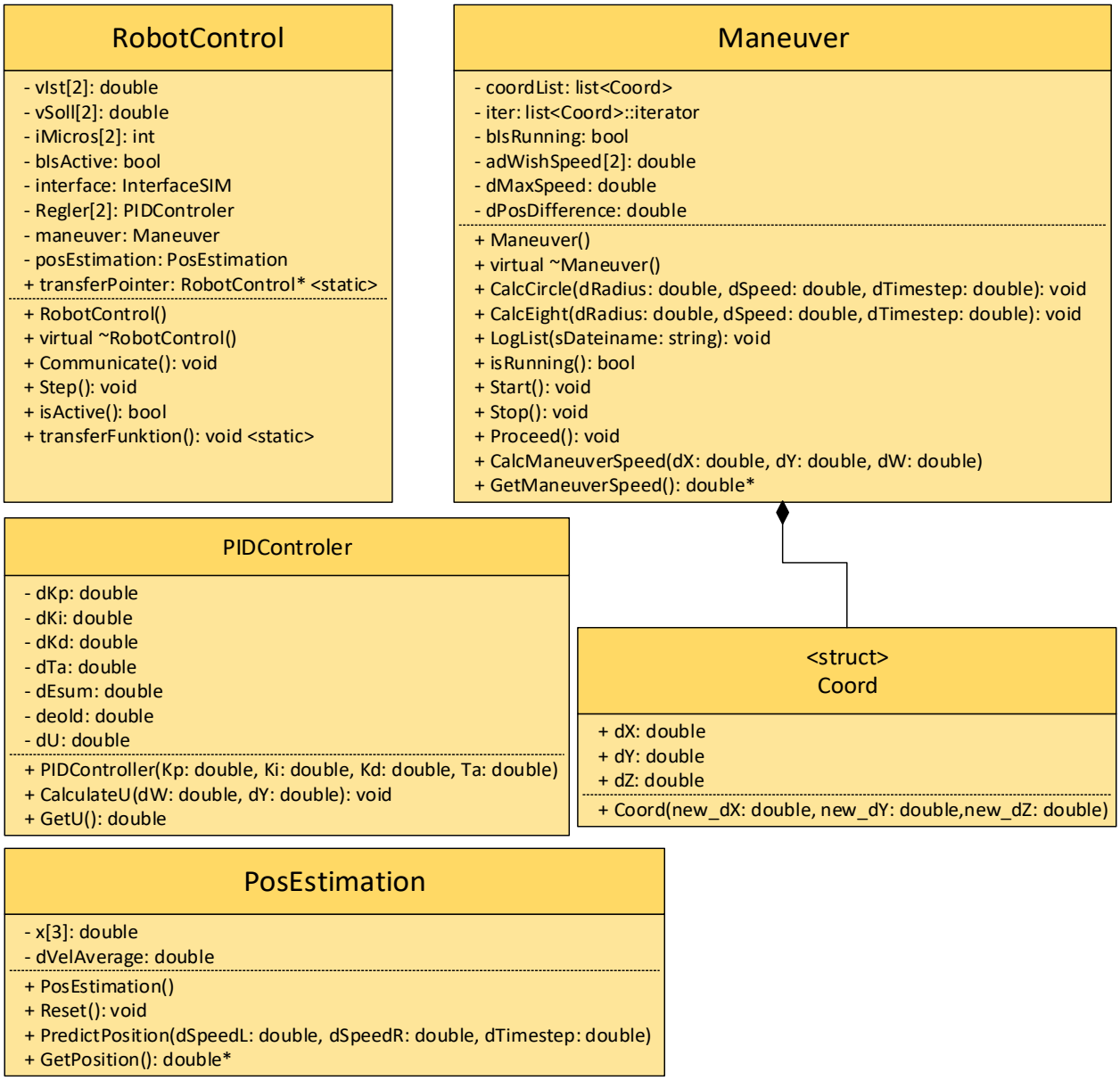
Präfix	Datentyp	Beispiel
i	integer	iSize
f	float	fZahl
b	boolean	bBusy
sz	null-terminierter String	szLastName
p	Zeiger	pMemory
a	Array	aCounter
ch	char	chName
r	Referenz	riSize

5.2.3 Präfixe der Sichtbarkeit

Um Variablen mit unterschiedlicher Sichtbarkeit, auch Scope genannt, leichter unterscheiden zu können, werden weitere Präfixe vorangestellt. Bei der objektorientierten Programmierung können damit Attribute einer Klasse von beispielsweise lokalen oder globalen Variablen sowie Funktionsübergabeparametern unterschieden werden.

Präfix	Sichtbarkeit	Beispiel
m_	Member-Variable	m_szLastName
p_	Methodenparameter	p_nNewValue
i_	Interfaceparameter (Argument von Funktionen)	i_nNewValue
s_	statische Variable	s_nInstanceCount
g_	globale Variable	g_nTimestamp

5.3 Klassendiagramme: Manöversteuerung Tag 5



“C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do it blows your whole leg off.”

— Bjarne Stroustrup

“Always code as if the guy who ends up maintaining
your code will be a violent psychopath who knows where you live”

— John Woods