Stand: 3. September 2017

Teil I
Einführung

Herzlich willkommen zum Übungsteil des Blockpraktikums `C++`!

1 Allgemeine Hinweise

Bevor Sie einsteigen, können Ihnen ein paar nützliche Tipps die Arbeit (hoffentlich) vereinfachen. Lesen Sie die Tipps durch, bevor Sie mit den eigentlichen Aufgaben beginnen!

Scheuen Sie sich desweiteren nicht, sich bei Unklarheiten, Fragen oder Wünschen an Michael Pehl oder die Tutoren zu wenden. Wir sind stetig bemüht das Praktikum Aktuell zu halten und zu Verbessern!

1.1 Nützliche Tipps

- Es wird des Öfteren passieren, dass Ihr Programm nicht die erwartete Ausgabe liefert. Überlegen Sie sich in solchen Fällen, welche **Zwischenergebnisse** Ihnen bei der Fehlersuche helfen könnten, und geben Sie diese Zwischenergebnisse aus. An manchen Stellen ist das Provozieren eines unerwarteten Verhaltens aber auch Teil der Aufgabenstellung, um Sie auf Fallstricke der Sprachen `C/C++` hinzuweisen, in diesem Fall sind die Hintergründe nach der Aufgabe erklärt.
- Halten Sie Ihre Programme **so einfach wie möglich**.
- Lesen Sie die Aufgaben vor dem Schreiben von Quellcodes vollständig durch und lösen Sie Unklarheiten bei den Aufgabenstellungen möglichst früh auf. Die Tutoren stehen Ihnen dazu gerne zur Verfügung.

1.2 Grundlegende Befehle der Kommandozeile

Manche Aufgaben setzen den Umgang mit der Linux-Kommandozeile voraus. Einen tiefen Einblick kann Ihnen hier nicht gegeben werden - die folgenden Befehle sind jedoch nützlich:

- `ls` zeigt den Inhalt des aktuellen Verzeichnisses an
- `cd <Neues-Verzeichnis>` wechselt das aktuelle Verzeichnis
- `mkdir <Verzeichnis>` legt <Verzeichnis> an
- `rmdir <Verzeichnis>` löscht <Verzeichnis>
- `rm <Datei>` löscht <Datei>

2 Kompilieren eines Programms

2.1 Texteditor oder Entwicklungsumgebung

Um Ihre Programme zum Laufen zu bekommen, müssen Sie diese zunächst in einem Texteditor oder einer Entwicklungsumgebung Ihrer Wahl schreiben. Ein empfehlenswerter Texteditor unter Linux und Windows ist **sublime text**. Alternativen für Linux sind

- emacs
- (g)vim
- nedit
- gedit
- kedit
- geany

Eine empfehlenswerte Entwicklungsumgebung (Integrated Development Environment, IDE) ist der QT-Creator, der speziell die Entwicklung von QT Applikationen unterstützen soll - dazu später mehr.

Ein wichtiger Unterschied zwischen einem Texteditor und einer IDE ist die Tatsache, dass bei Letzterer bestimmte Prozessschritte (Kompilieren, Debuggen, Ausführen) bereits in die grafische Benutzeroberfläche integriert sind. Trotzdem ist es wichtig, dass Sie die einzelnen Schritte verstehen, weshalb wir Ihnen nahelegen möchten die ersten Aufgaben mit einem Texteditor und Commandline Werkzeugen zu lösen.

2.2 Compiler

Der Build-Prozess wird in einer späteren Übungsaufgabe detailliert behandelt. Verwenden Sie daher zunächst folgende Befehle zum Erstellen des Programms, die Sie in ein Terminal (z.B. `gnome-terminal` oder `xterm` eintippen).

```
<compiler> -o<executable> <source>
./<executable>
```

Die erste Zeile kompiliert die Datei `<source>` und erstellt das ausführbare Programm unter dem Dateinamen `<executable>`. Die zweite Zeile führt `<executable>` schließlich aus. `<compiler>` steht dabei für

- `g++`, wenn Sie `C++`-Programme kompilieren wollen
- `gcc`, wenn Sie `C`-Programme kompilieren wollen
- Andere Compiler, je nach Sprache oder Compiler Suite

Wenn Sie Ihre Sourcedatei `mySampleProgram.cpp` also mit dem `g++` Compiler aus der GNU Compiler Collection (GCC) und dem `C++` Standard 14 kompilieren wollen, könnte das benötigte Command dafür wie folgt aussehen:

```
g++ -std=c++14 mySampleProgram.cpp -o mySampleProgram
./mySampleProgram
```

Nun sollten Sie bereit sein, die Übungsaufgaben zu bearbeiten.

Teil II
Aufgaben

3 Aufgaben für Tag 1

3.1 Basics: Variablen und Funktionen

In der Vorlesung haben Sie Datentypen und Funktionen kennen gelernt. Hier sollen Sie Ihr Wissen anwenden, allerdings brauchen Sie noch ein paar zusätzliche Informationen um „spannendere“ Programme schreiben zu können.

3.1.1 Ausgabestreams in C++

Standardmäßig wird die Ausgabe in C++ über die Outputstreams `std::cout` und `std::cerr` verwirklicht. `std::cout` schreibt dabei auf den Standardoutputkanal[1] und `std::cerr` auf den StandardFehlerkanal[2], die Streams können also sowohl als Output als auch als Fehlermeldung *an den User* genutzt werden. Um die Streams `std::cout` und `std::cin` zu benutzen muss die `iostream` Bibliothek eingebunden (inkludiert) werden.

Mit Outputstreams wird Output mit dem Linksschiebeoperator (Streamoperator) `<<` nach `stdout` und damit auf den Bildschirm *gestreamed*.
Beispiele:

```
1 std::cout << "Streamtest" << std::endl;
2
3 int someNumber = 42;
4 std::cout << someNumber << std::endl;
5 std::cout << "Some␣Number␣is:␣" << someNumber << std::endl;
```

Das Hello-World Programm kann in C++ also wie folgt implementiert werden:
`helloworld.cpp`

```
1 #include <iostream>
2
3 int main(int, char **) {
4
5     std::cout << "Hello␣World" << std::endl;
6
7     return 0;
8 }
```

Hinweis: Die `main`-Funktion nimmt Eingabeparameter von der Command Line (CL), also der Konsole/dem Terminal in Linux, an. werden als Integer `argumentcount`, bzw, `argc`, und Textarray[3] `argumentvalue`, bzw. `argv`, übergeben. Benötigt man die Argumente nicht, weil man z.B. kein Command Line Interface (CLI) implementiert, empfiehlt es sich die Argumente trotzdem (ohne Namen) anzugeben, man kann diese aber auch komplett weg lassen.

[1] In Linux `stdout`
[2] In Linux `stderr`
[3] dazu in der Vorlesung später mehr

3.1.2 Ausgabe mit Variablen

In Ihrer ersten Aufgabe sollen Sie sich mit Variablen und Funktionen vertraut machen. Dafür sollen Sie zunächst ein einfaches Programm schreiben und dieses nach und nach modularisieren.

1. Erstellen Sie eine neue Textdatei `name.cpp` und adaptieren Sie das obige Beispiel und lassen Sie Ihren/ein Namen und Ihr/ein Alter ausgeben.
   Beispielausgabe: `Tom Sawyer - 12 Jahre`
2. Führen Sie Variablen ein, die jeweils den Namen und das Alter speichern und lassen Sie den Namen und das Alter mit den Variablen ausgeben. Wählen Sie angemessene Datentypen für die jeweiligen Variablen.
3. Führen Sie eine Funktion `printNameAndAge` ein, die den Namen und das Alter als Parameter annimmt und auf der Command Line ausgibt. Die Funktion muss nichts zurückgeben.

Hinweis: Wenn Sie `std::string` benzutzen möchten, müssen Sie zusätzlich die `string` Bibliothek inkludieren.

3.2 Hallo sagen

Als nächstes soll `name.cpp` so erweitert werden, dass nach dem Name und dem Alter des Benutzters gefragt wird. Anschließend soll, wie zuvor, beides ausgegeben werden und zusätzlich das Alter des Benutzers kommentiert werden. Zum Beispiel könnte ein sehr junger Benutzer mit „Jungspund“ und das Alter eines Seniors mit „Du bist aber alt“ kommentiert werden.

3.2.1 Eingabestreams in `C++`

Analog zu den Outputstreams wir in `C++` standardmäßig mit Hilfe des Inputstreams `std::cin` in eine Variable eingelesen. `std::cin` ist in `iostream` definiert.

```
std::cin >> variablenName;
```

Um also beispielsweise eine Zahl einzulesen:

```
1 int someNumber;
2 std::cin >> someNumber;
```

3.2.2

Fragen Sie nun den Nutzer nach Name und Alter und reagieren Sie entsprechend. Gehen Sie dabei wie folgt vor:

1. Erweitern Sie `name.cpp`, sodass der User nach seinem Namen und Alter gefragt wird.
2. Erweitern Sie die Ausgabe so, dass verschiedene Altersgruppen anders angesprochen werden.

3.3 Ein kleines Spiel

In dieser Aufgabe sollen Sie ein kleines „Guess the Number“ Spiel schreiben. Dabei spielen zwei Spieler gegeneinander, wobei ein Spieler *Spieler1* eine zufällige Zahl wählt und der andere Spieler *Spieler2* versucht die Zahl zu erraten. *Spieler1* gibt jedes mal Feedback, ob *Spieler2* die Zahl

erraten hat oder zu hoch, bzw zu niedrig getippt hat. In diesem Fall soll der Computer *Spieler1* sein.

3.3.1 Zufallszahlen

Um eine einfache Zufallszahl zu erzeugen können Sie `std::rand` benutzen[4]. `std::rand` erzeugt eine pseudo-random Number, das heißt generierte Zahlen werden deterministisch generiert. Um also jedes mal eine neue Zahl zu erzeugen müssen Sie diesen Generator *seeden*, also am besten mit einer Zufallszahl initialisieren. Dies funktioniert am einfachsten mit `std::srand`. Als Seed benutzen Sie die aktuelle Zeit `std::time(0)`. Insgesamt müssen Sie also zusätzlich die Bibliotheken `random` und `chrono` (für den Seed) inkludieren:

```
1 #include <random>
2 #include <chrono>
3
4 /* ... */
5 std::srand(std::time(0));
6 int someRandomNumber = std::rand();
```

3.3.2

Um das Spiel nun zu implementieren, gehen Sie wie folgt vor:

1. Erstellen Sie eine neue Textdatei numberGame.cpp und definieren Sie die Funktionen `getNumber` und `playGame`.

`getNumber` soll sowohl die kleinstmögliche als auch die größtmögliche positive Zahl als Parameter annehmen. Als Standardargumente können Sie `0`, bzw. `std::numeric_limits<int>::max()` aus der `limits` Bibliothek benutzen. Die Funktion soll die generierte Zahl zurück geben.

`playGame` soll ein Argument, die zufällig generierte Zahl, annehmen.

2. Fügen Sie den Code hinzu um eine Zufällige positive Zahl zu erzeugen.
   - Seeden Sie dazu zunächst den Zufallszahlengenerator in der `main`-Funktion.
   - Generieren Sie in `getNumber` eine Zufallszahl und geben Sie sie zurück.
3. Fügen Sie den Code hinzu um Input vom User einzulesen und mit der Zufallszahl zu vergleichen
   - In der Funktion `playGame`, lesen Sie den Tipp des Users in eine Variable `guess` ein.
   - Vergleichen Sie den Input mit der als Parameter übergebenen Zahl und geben Sie dem User entsprechend Output.
   - Lassen Sie den User so lange tippen, bis er die richtige Zahl errät.

[4]Später werden Sie noch eine weitere, bessere Alternative lernen Zufallszahlen zu erzeugen. Für diese Aufgabe wird diese Möglichkeit jedoch ausreichen.

3.4 Build-Prozess

Die Abfolge von Aktionen, um aus Quellcodes und Bibliotheken ein lauffähiges Programm („executable“) zu erzeugen, wird Build-Prozess genannt. An dieser Stelle sollen Sie den Build-Prozess näher kennen lernen, der dafür sorgt, dass aus den Quellcodes und Bibliotheken eine ausführbare Datei erzeugt wird. Benutzen Sie für die folgenden Aufgaben den gegebenen Quellcode des `name_age` Projektes.

3.4.1 Manuelles Erstellen mit `g++`

Das manuelle kompilieren wurde bereits angesprochen, allerdings soll es hier um das „stückweise“ kompilieren eines Projektes gehen, um bei großen Projekten die Kompilierzeiten zu verkürzen. Verwenden Sie zunächst den `C++`-Compiler `g++` aus der GNU Compiler Collection, um zunächst alle Dateien zu **kompilieren**, d.h. Objektcodes zu erzeugen. Wie aus der Vorlesung bekannt, erzeugt dieser Schritt bereits maschinenlesbaren Code, der jedoch noch keine Adressen enthält. Zu diesem Zeitpunkt ist nämlich noch nicht bekannt, aus welchen Komponenten die endgültige ausführbare Datei zusammengesetzt sein wird.

Um die Objektdatei *module.o* aus der Quelldatei *module.c*pp zu erstellen, wird `g++` wie folgt aufgerufen:

```
g++ -c module.cpp
```

Wenn alle Objektcodes erstellt sind, müssen die Dateien **gelinkt** werden. Auch dazu kann `g++` verwendet werden:

```
g++ -o executable module_1.o module_2.o module_3.o
```

erzeugt die Programmdatei `executable` aus den Objektcodes `module_1.o`, `module_3.o` und `module_3.o`.

Erzeugen Sie nun das `name_age` Projekt:

1. Erstellen Sie aus den Dateien `name_age.cpp` und `main.cpp` den Objektcode.
2. Linken Sie die Objektcodes zu einem ausführbaren Programm `name_age`.
3. Führen Sie das soeben erstellte Programm `name_age` aus und testen dessen korrekte Funktionalität.

3.4.2 Automatisierte Erstellung mittels `make`

Für Programme mit vielen Quelldateien ist das Kompilieren/Linken mittels eigenständiger Compiler-Aufrufe zu aufwändig. Die Verwendung von `make` ist der erste Schritt zur Automatisierung des Build-Prozesses. In einem `Makefile` sind so genannte *Make-Targets* definiert. Sie beinhalten:

- **Name** des Make-Targets
- **Abhängigkeiten** von anderen Make-Targets
- Anweisungen zum **Erstellen** des Make-Targets

Ein Make-Target ist entweder eine **Datei**, die als Zwischenschritt im Build-Prozess benötigt wird (z.B. `main.o`), oder eine Operation, die ausgeführt werden soll (z.B. `install`). Solche Operationen oder Rezepte, können mit `.PHONY` als explizite Anforderung definiert werden, was die

Performance erhöhen kann. Es sollte aber darauf geachtet werden, dass Dateien mit dem Namen eines `.PHONY` Targets nie existieren!

Der Aufbau einer Make-Target-Definition ist die folgt:

```
<Name des Make-Targets>: <Abhaengigkeiten>
<Tabulator><Anweisung zum Erstellen des Make-Targets>
```

Beachten Sie, dass die Einrückung der Anweisungen zum Erstellen der Make-Targets durch **Tabulatoren**, und **nicht** durch **Leerzeichen** erfolgt. Die so angegebenen Anweisungen werden von `make` blind ausgeführt. Der Programmierer muss die Anweisungen so wählen, dass das Make-Target auch tatsächlich wie gewünscht erstellt wird.

Ein Beispiel-`Makefile`, das zu einem `C`-Projekt mit der Datei `main.cpp` besteht, könnte wie folgt aussehen.

```
executable: main.o
        g++ -o executable main.o

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

.PHONY: install
install: executable
        cp ./executable /bin
        chmod 555 /bin/executable
        chown root:root /bin/executable

.PHONY: clean
clean:
        rm *.o
        rm executable
```

Hier sind vier Make-Targets definiert:

Make-Target	Abhängigkeiten	Anweisungen
`executable`	`main.o`	**Linken**
`main.o`	`main.cpp`	**Kompilieren**
`install`	`executable`	Kopiert Programm dorthin, wo Programme „normalerweise“ sind
`clean`	–	Entfernt alles, was im Build-Prozess erzeugt wurde

Durch den Aufruf von `make <Target>` im Verzeichnis, in dem sich `Makefile` befindet, wird das gewünschte Target erstellt, `make executable` erstellt also das Target `executable`. Durch die Angabe der Abhängigkeiten kann das Programm `make` prüfen, welche anderen Targets erstellt werden müssen. Dabei erkennt `make` geänderte Dateien im Abhängigkeitsgraph automatisch über den Zeitstempel: ist also beispielsweise `main.cpp` neuer als `main.o`, so wird `main.o` beim Aufruf von `make testprogramm` automatisch auch neu erstellt. Beim Aufruf von `make` ohne Target wird das erste Target (Defaulttarget) ausgeführt.

Erstellen Sie nun ein `Makefile`, die das Binary `name_age` aus den Quellen `name_age.cpp` und `main.cpp` automatisch erstellt. Folgende Targets sollen existieren

- `name_age`

- `name_age.o`
- `main.o`

Die Targets `install` und `clean` sind nicht nötig.

1. Überlegen Sie sich zunächst, von welchem/n anderen Target(s) die drei Targets unmittelbar abhängen.
2. Schreiben Sie die Abhängigkeiten in eine Datei namens `Makefile` im selben Verzeichnis wie die Quellcodes. Orientieren Sie sich dabei am oben genannten Beispiel.
3. Tragen Sie für die Make-Targets die Befehle ein, die zum Erstellen nötig sind.
   Achtung: Solche Zeilen müssen mit einem **Tabulator** beginnen, danach muss unmittelbar die Befehlszeile folgen.
4. Prüfen Sie mittels des Befehls `make name_age` die korrekte Funktion Ihrer Makefile.
5. Machen Sie nun eine Änderung in der Datei `main.cpp` (z.B. Hinzufügen einer Leerzeile). Rufen Sie anschließend `make name_age` erneut auf. Welche Build-Schritte werden nun ausgeführt?

3.5 Vollautomatischer Build-Prozess mit `cmake`

Den Build-Prozess über Abhängigkeiten einzelner Dateien zu beschreiben ist spätestens bei größeren Projekten sehr zeitaufwändig. Zudem muss man bei Abhängigkeiten von externen Bibliotheken u.U. konkrete Verweise zu Bibliotheksdateien angeben und verliert damit einen gewissen Grad an Plattformunabhängigkeit verlieren.

Deshalb ist es sinnvoll, die Abhängigkeiten auf einer höheren Abstraktionsebene zu beschreiben und den Build-Prozess weiter zu automatisieren. Dazu gibt es verschiedene Build-Systeme; im Unix-Bereich hat `cmake` eine führende Stellung eingenommen. `cmake` kann aber aber auch unter Windows verwendet werden.

Für den Praxisteil des `C++`-Praktikums kann CMake eingesetzt werden. Hier sollen Sie lernen, die Grundlagen von CMake zu verstehen, indem Sie die Build-Umgebung für das Erstellen von `name_age` verwenden.

CMake verlangt im Quellverzeichnis eine Datei namens `CMakeLists.txt`. Hier werden alle für den Build-Prozess nötigen Optionen angegeben; bei einem größeren Projekt sind dies beispielsweise:

- die CMaken Version
- der Projektname
- der zu verwendende `C++` Standard (für dieses Praktikum mindestens 11, gewöhnlich 14)
- die Dateinamen der zu erstellenden Programme, und dafür nötige Quelldateien
- die Dateinamen der zu erstellenden Bibliotheken, und dafür nötige Quelldateien
- Abhängigkeiten von externen Bibliotheken, ggf. mit Angabe zur benötigten Version
- bei Qt: Angabe von Header-Files, die eine speizelle Behandlung erfordern (dazu später mehr)

Für ein Projekt, welches nur die Quelldatei `main.cpp` enthält, sieht eine die Datei beispielsweise wie folgt aus. Diese Datei weist das Build-System an, die Programmdatei `executable` aus der Quelldatei `main.cpp` zu erstellen.

```
cmake_minimum_required(VERSION 2.8.11)
project(mySampleProject)
set(CMAKE_CXX_STANDARD 14)
add_executable(executable main.cpp)
```

CMake erlaubt es, die beim Build-Prozess entstehenden Dateien (Objektcodes, ...) außerhalb des Build-Verzeichnisses zu erstellen. Dies sorgt für eine bessere Übersichtlichkeit. Die Version (2.8.11) ist hier zufällig gewählt, allerdings kann es sein, dass verschiedene CMake Commands oder Projekte (z.B. Qt basierende) eine bestimmte Mindestversion erfordern.

Um den Build-Prozess zu starten, sind folgende Schritte nötig:

1. Erstellen des Verzeichnisses, in dem die Build-Dateien abgelegt werden sollen (z.B. `build`)
2. Wechsel in dieses Verzeichnis
3. Aufruf von `cmake` mit dem Quellcode-Verzeichnis als einzigem Parameter (z.B. `cmake ../src`). `cmake` erstellt dadurch ein `Makefile`.
4. Aufruf von `make` ohne weitere Parameter.

Folgendes Listing soll das Beispiel eines Build-Prozesses zeigen:

```
user@Compi:~/testproject$ mkdir build

user@Compi:~/testprogram$ ls -l
total 8
drwx------ 2 user ugroup 4096 2012-10-15 16:05 build
drwx------ 3 user ugroup 4096 2012-10-15 16:03 src
user@Compi:~/testprogram$ cd build

user@Compi:~/testprogram/build$ cmake ../src
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/user/testprogram/build

user@Compi:~/testprogram/build$ make
Scanning dependencies of target executable
[100%] Building CXX object CMakeFiles/executable.dir/main.cpp.o
Linking CXX executable executable
[100%] Built target executable

user@Compi:~/testprogram/build$ ./executable
Hallo Welt!
```

Man sieht, dass CMake beispielsweise überprüft, ob ein für das Projekt passender Compiler vorhanden ist. Gerade in größeren Projekten hängt man nicht nur vom Compiler ab, sondern z.B. auch von Programmbibliotheken, oder man benötigt Fallunterscheidungen in Abhängigkeit des Betriebssystems.

Ein tieferer Einblick in CMake würde den Rahmen dieses Übungsteils sprengen, bei Interesse finden Sie beispielsweise unter `http://www.cmake.org/cmake/help/cmake_tutorial.html` weitere Informationen.

Schreiben Sie nun eine `CMakeLists.txt` Datei, die das `name_age` Programm kompiliert.

1. Erstellen Sie eine `CMakeLists.txt` im Verzeichnis, in dem sich auch die `C++`-Quellcodes befinden. Übergeben Sie `add_executable(...)` sowohl `name_age.cpp` als auch `main.cpp`, denn beide Dateien sollen zu einem Programm gelinkt werden.
2. Erzeugen Sie ein Build-Verzeichnis
3. Erstellen Sie `name_age` im Build-Verzeichnis
4. Führen Sie `name_age` aus und überprüfen Sie dessen korrekte Funktion.

3.6 FizzBuzz

Als letzte Aufgabe sollen Sie FizzBuzz implementieren. Dabei sollen Sie von `1` bis zu einer beliebigen Zahl zählen und dabei Zahlen, die durch 3 teilbar sind, durch `Fizz` ersetzen, und Zahlen die durch 5 teilbar sind, durch `Buzz` ersetzen. Zahlen, die sowohl durch 3, als auch durch 5 teilbar sind, sollen durch `FizzBuzz` ersetzt werden.

1. Erstellen Sie die neuen Dateien `fizzbuzz.cpp` und `fizzbuzz.hpp`.
2. Deklarieren Sie in `fizzbuzz.hpp` eine Funktion `fizzbuzz`, die eine Zahl als Parameter annimmt.
3. Fügen Sie die Implementierung von `fizzbuzz` hinzu, sodass alle einschließlich des übergebenen Parameter oder entsprechend `FizzBuzz`/`Fizz`/`Buzz` Zeile für Zeile ausgegeben werden.
4. Testen Sie ihr Programm. Schreiben Sie dazu entweder eine eigene `main` Funktion in einer neuen Datei `main.cpp`, oder nutzen Sie die gegebene `main` Funktion in `main.o`.
5. Schreiben Sie eine Projektdatei für das Buildsystem Ihrer Wahl (Make oder CMake) und kompilieren Sie das Projekt damit.

4 Aufgaben für Tag 2

4.1 Enums

In der folgenden Aufgabe werden Sie die Vor- und Nachteile kennenlernen, die `enum class`es mit sich bringen.

1. Definieren Sie in Ihrer Datei `enum.cpp` einen normalen `enum`-Typ namens `ShirtColor`, der folgende Werte annehmen kann:

   - `RED`
   - `BLACK`
   - `WHITE`

   Legen Sie nun eine Variable des Typs `ShirtColor` mit dem Namen `shirtColor` in Ihrer `main`-Funktion an und initialisieren Sie diese mit `RED`. Weisen Sie Ihr Programm als nächstes dazu an, eine Fallunterscheidung per `switch` vorzunehmen, um dann je nach Inhalt der Variable die richtige Farbe auszugeben.

2. Fügen Sie nun zusätzlich die Farbe `YELLOW` vor `RED` in die `enum`-Definition ein und stellen Sie die korrekte Verhaltensweise Ihres Programmes sicher.

3. Legen Sie nun einen zweiten `enum`-Typ an, der den Namen `FlowerColor` tragen soll. Dieser soll folgende Werte annehmen können:

   - `ROSE_RED`
   - `LILAC`
   - `SUN_YELLOW`
   - `WHITE`

   Versuchen Sie anschließend das Programm zu kompilieren. Was stellen Sie fest?

4. Dieses Problem kann umgangen werden, indem man jedes `enum` in einen eigenen Namensraum (engl.: `Namespace`) verschiebt. Verifizieren Sie diese Aussage.

5. Leider ist auch diese Lösung nicht ohne Makel, denn der Namensraum kann an jeder Stelle im Quelltext beliebig erweitert werden, so dass auch weitere Kollisionen nicht ausgeschlossen sind. Deswegen wurde mit dem `C++`11-Standard die Möglichkeit eingeführt, `enum class`es zu erstellen. Testen Sie nun Ihr Programm mit diesen neuen Konstruktionen.

4.2 Referenzen

Oft will man einer Funktion Variablen zu Bearbeitung übergeben, wobei die Variable selbst dann geändert werden soll. Mit der Übergabe von normalen Variablen (`call by value`) wird lediglich der Wert einer Variable, nicht aber deren Speicheradresse übergeben. Die Lösung ist, Variablen als Referenzen zu übergeben (`call by reference`). Dabei wird nicht der Wert einer Variable, sondern die Speicheradresse übergeben und der Eingabeparameter kann somit als *Alias* zu der Originalvariable genutzt werden.

Eine von Ihnen zu implementierende Funktion `swapInt` soll zwei zu übergebende Zahlen vertauschen.

1. Implementieren Sie in der Datei `swap.cpp` die Funktion `swapInt`, welche zwei zu übergebende Zahlen des Typs `int` vertauscht. Die Deklaration der Funktion soll wie folgt aussehen:
   `void swapInt(int &, int &);`
2. Implementieren Sie in der selben Datei nun auch eine `main()`-Funktion.
   (a) Legen Sie zwei `int`-Variablen an und initialisieren Sie diese mit den Werten `23` und `42`.
   (b) Geben Sie beiden Variablen mittels `cout` auf dem Bildschirm aus.
   (c) Vertauschen Sie den Inhalt der beiden Variablen, indem Sie die Funktion `swapInt` aufrufen.
   (d) Geben Sie die beiden Variablen erneut aus und überprüfen Sie so, ob die `swapInt`-Funktion korrekt implementiert ist.

4.3 Klassen

An dieser Stelle sollen Sie Ihre erste eigene Klasse implementieren.

In `C++` bestehen Quellcodes aus Headerdateien (empfohlene Endung `.hpp`) und Quelldateien (empfohlene Endung `.cpp`). Wie bereits zu erwarten ist, enthält eine `C++`-Headerdatei Deklarationen und die Quelldatei Implementationen. Für Klassen kommen die Deklaration der Klasse und ihrer Methoden, sowie die Implementation der Methoden hinzu.

4.3.1 Beispielaufgabe

Als einfaches Beispiel soll eine Klasse *Car* implementiert werden. Die Struktur der Klasse soll wie folgt aussehen:

- Unveränderliche Merkmale einer Car-Instanz
  - Kapazität des Tanks in Litern
  - Benzinverbrauch in Litern pro 100 km
- Zustandsabhängige Eigenschaften
  - Füllstand des Tanks
  - Kilometerstand

Folgende Methoden sollen unterstützt werden:

- **refuel.** Als Parameter erwartet werden soll die zu tankende Benzinmenge in Litern (`double`). Die Operation soll nur dann ausgeführt werden, wenn genügend Platz im Tank vorhanden ist. Der Erfolg der Operation soll als `bool`-Rückgabewert berichtet werden.
- **drive.** Als Parameter erwartet werden soll die zurückzulegende Strecke (`double`). Beachten Sie, dass das Auto natürlich nur so lange fahren kann, so lange genügend Benzin im Tank vorhanden ist. Der Rückgabewert soll die tatsächlich zurückgelegte Strecke als `double` enthalten.
- **fillLevel** soll den aktuellen Füllstand als `double` zurückgeben

- **reach** soll die aktuelle Reichweite als `double` zurückgeben
- **milage** soll den aktuellen Kilometerstand als `double` zurückgeben

Ein Testprogramm soll schließlich folgende Testfälle abarbeiten:

1. Erzeugen des Autos und Übergabe der unveränderlichen Merkmale (50 l, 5 l/100km) im Konstruktor
2. Tanken von 20l
3. Füllstand, Kilometerstand und Reichweite ausgeben
4. Fahren von 10km
5. Füllstand, Kilometerstand und Reichweite ausgeben
6. Volltanken
7. Füllstand, Kilometerstand und Reichweite ausgeben
8. Fahren, bis Tank leer ist
9. Füllstand, Kilometerstand und Reichweite ausgeben
10. Testen, ob Weiterfahren auch wirklich fehlschlägt
11. Füllstand, Kilometerstand und Reichweite ausgeben
12. Volltanken
13. Füllstand, Kilometerstand und Reichweite ausgeben
14. Versuchen, 10 l zusätzlich zu tanken.
15. Füllstand, Kilometerstand und Reichweite ausgeben

1. Erstellen Sie ein UML-Diagramm der zu erstellenden `Car`-Klasse.
2. Implementieren Sie die Klasse in die Dateien `car.hpp` und `car.cpp`. Um zu verhindern, dass die unveränderlichen Merkmale versehentlich überschrieben werden können, sollen diese als konstant (`const`) deklariert werden. Auch die Methoden, die den Zustand der Objekte nicht ändern, sollen als `const` deklariert sein.
3. Implementieren Sie das Testprogramm unter dem Namen `cartest.cpp`.
4. Führen Sie das Testprogramm aus, bewerten Sie das Ergebnis und korrigieren Sie eventuell auftretende Fehler.

5 Aufgaben für Tag 3

5.1 Zeiger

5.1.1 Einführung

Vor Referenzen waren Zeiger (Pointer) die einzige Möglichkeit Speicheradressen weiter zu geben oder „Aliase“ zu erzeugen. Und auch mit Referenzen bleiben Pointer einer der mächtigsten und wichtigsten Mechanismen von `C++`.

Vor Referenzen war ein Anwendungsfall von Pointern die `call by reference` Implementierung von Funktionen. Dieselbe Funktionalität soll nun von Ihnen mit Pointern verwirklicht werden. Das heißt, dass die vorherige `swapInt` Funktion nun mit Pointern anstatt Referenzen verwirklicht werden soll.

1. Implementieren Sie in der Datei `swap.cpp` die Funktion `swapInt`, welche zwei zu übergebende Zahlen des Typs `int` vertauscht. Die Deklaration der Funktion soll wie folgt aussehen:
   ```
   void swapInt(int *, int *);
   ```

2. Rufen Sie nun in der `main`-Funktion `swapInt` auf. Vergessen Sie dabei nicht, dass Sie den Pointer auf Ihre Variablen übergeben müssen. Geben Sie die Variablen anschließend aus und überprüfen Sie die Funktionalität.

3. Kompiliert Ihr Programm, wenn Sie die neue `swapInt` Funktion in der selben Datei in der Sie auch die Referenz-`swapInt` Funktion implementiert haben? Wenn ja, wieso? Wenn nein, wieso nicht?

4. Wie müsste die Deklaration einer Funktion `swapIntPointer` aussehen, die, statt den `int`-Werten selbst, Zeiger auf `int`-Werte (d.h. `int*`-Typen) vertauscht? Ändert sich dafür die Implementierung?

Hinweis: Derzeit ist Konsens, raw Pointer so weit wie möglich zu vermeiden und stattdessen Smartpointer oder Referenzen zu benutzen. Dies ist in manchen Fällen nicht möglich, sodass auch raw Pointer immer noch ein wichtiges und mächtiges Werkzeug in `C++` bleiben! In obiger Aufgabe ist allerdings offensichtlich die Implementierung mit Referenzen vorzuziehen.

5.2 Heap-Verwaltung mit `new` und `delete`

In der Vorlesung haben Sie den Unterschied zwischen Heap und Stack kennen gelernt. Hier sollen Sie nun mit den Operatoren `new` Objekte auf dem Heap anlegen, benutzen und anschließend den Speicher mit `delete` wieder freigeben.

1. Erstellen Sie ein Programm `cartest2.cpp`, dass `Car` mittels `new`-Operator auf dem Heap erzeugt. Rufen Sie eine Methode auf (dazu benötigen Sie nun den `->`-Operator). Vergessen Sie nicht, das Objekt am Ende des Programmes mit `delete` zu zerstören und den Speicher freizugeben.

2. Kompilieren Sie das Programm und führen Sie es aus.

5.3 Templates

Wie Sie aus der Vorlesung wissen, sind Templates Baupläne für Funktionen und Klassen, die für verschiedene Datentypen verwendet werden, deren Implementierung ansonsten aber gleich ist.

Eine Template-Klasse wird, wie bereits erwähnt, wie folgt deklariert:

```
template<class T>
class MyClass {
    private:
    // Example: some member of type T
    T _someMember;

public:
    // Example: Method taking T& as parameter
    void processData(T& data);
};
```

Im oben genannten Beispiel kann nun `T` innerhalb der Klassendeklaration als Platzhalter für einen Variablentyp verwendet werden.

Die Implementierung der Template-Klassen-Methoden erfolgt in der Headerdatei (!!) wie folgt:

```
template<class T>
void MyClass<T>::processData(T& data) {
    // Implementation
}
```

Instantiiert wird eine Template-Klasse dann wie folgt:

```
#include "myClass.hpp"

int main() {
    MyClass<double> myObject;
    // ...
    myObject.processData(5.0);
}
```

Templates sind hervorragend zur Implementierung von Datencontainern geeignet: Dessen Operationen (Anlegen eines Elements, Schreiben eines Elements, Lesen eines Elements, Löschen eines Elements) sind – bis auf den verwendeten Datentyp – vom Code her identisch.

5.3.1 Verkettete Liste (Linked List)

Was ist eine Linked List? Einfach verkettete Listen basieren auf Datenstrukturen, dessen Instanzen miteinander verkettet werden. Die Datenstruktur, die im Folgenden Listenelement genannt wird, enthält folgende Elemente:

- Ein zu speicherndes „Nutzdaten“-Element (z.B. ein `int`-Wert oder ein `std::string`).
- Ein Verweis (Pointer) auf das unmittelbar folgende Listenelement.

Ein Beispiel einer einfach verketteten Liste ist in Abbildung 1 gezeigt. Linked Lists sind damit eine Möglichkeit viele Daten zu speichern, ohne zur Programmierzeit wissen zu müssen, wie viele Daten wohin gespeichert werden. Um eine Linked List selbst zu implementieren braucht man mindestens 2 Sachen.

1. Eine Struktur (Klasse), die ein Element der Liste enthält (`ListElement`)

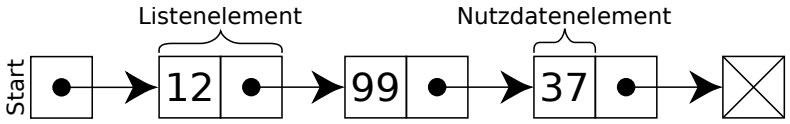

Abbildung 1: Beispiel einer einfach verketteten Liste

2. Einen Supervisor (Start) der Liste, der weitere Informationen über die Liste enthält, aber nicht muss (`List`, im einfachsten Fall kann der Kopf der Liste sogar das erste Listenelement sein)

Das Listenelement besteht im einfachsten Fall aus einem Datum und dem Pointer auf das nächste Element und implementiert zusätzlich alle Methoden, die gebraucht werden um ein Element einzufügen, zu entfernen oder zu adressieren (hängt immer auch von der gewählten Implementierung ab). Der Supervisor hat normalerweise die Aufgabe die Liste zu verwalten und bietet damit ein Interface um Daten einzufügen, zu entfernen oder zu adressieren. Ausserdem hat der Supervisor auch oft weitere Informationen wie die Länge der Liste oder bietet die Möglichkeit die Liste zu sortieren, ...

Datenstrukturen Die interne Struktur eines Listenelements `ListElement` soll durch eine Klasse realisiert werden. Das tatsächlich gespeicherte Datum soll dabei als Template-Argument übergeben werden. `ListElement` soll aus mindestens zwei Elementen bestehen:

- Dem Datum vom übergebenen Templatedatentyp
- Dem Pointer auf das nächste Element
  Wählen Sie dafür eine geeignete Pointerimplementation (`std::shared_ptr` oder `std::unique_ptr`)

Die interne Struktur des Supervisors `List` soll ebenfalls durch eine Klasse realisiert werden. Die Klasse soll dabei mindestens den Pointer auf das erste Element, also den Listenkopf enthalten (wählen Sie auch hier eine angemessene Implementierung). Zudem soll `List` folgende Schnittstellen als Methoden bieten.

Schnittstellen

- Konstruktion einer leeren Liste (Konstruktor `List`)
- `T& at(const std::size_t position)`
  Soll die Referenz auf das Datum an gegebener Position zurück geben.
- `void insertItem(const T &data, const std::size_t position)`
  Soll ein Element `data` vor der `position`-ten oder an der letzten Position in die Liste einfügen.
- `void deleteItem(const std::size_t position)`
  entfernt das `position`-te Listenelement aus der Liste.
- `const size_t count()`
  Soll die aktuelle Anzahl an Elementen in der Liste zurückgeben.

Das erste Element der Liste soll dabei den Index `0` haben.

Kapselung von `ListElement` `C++` erlaubt es Klassen innerhalb von Klassen zu Deklarieren. Wenn Sie also die genaue Implementierung Ihrer Liste nach aussen verstecken wollen, können Sie `ListElement` in die `List` Klasse verschieben.

Fehlerbehandlung Überlegen Sie sich, wie sie mögliche Fehler abfangen können. Ein möglicher Fehler ist in erster Linie, die Anfrage einer ungültigen Position. Ein möglicher Workaround (bis später) ist, eine Variable in `List` einzuführen, die die Gültigkeit des übergebenen Elements angibt und im Zweifel eine Dummyvariable zurück zu geben. Aber auch das stößt hier schnell an Grenzen. Finden Sie heraus welche?
Anmerkung: Beachten Sie, dass das leere Element, welches im Fehlerfall ausgeliefert wird, nicht kürzer als das `List<T>`-Objekt gültig sein darf. Dies können Sie beispielsweise erreichen, indem Sie das Dummyobjekt als Member-Variable von `List<T>` vorhalten. Nicht gültig wäre zum Beispiel das Liefern einer Referenz auf eine lokale Variable von `at`.

1. Erstellen Sie eine Datei namens `linkedlist.hpp` und deklarieren Sie dort die Klassen `List` und `ListElement` mit den oben angegebenen Eigenschaften und Methoden.

2. Implementieren Sie die Methoden ebenfalls in der Datei `linkedlist.hpp` oder `linkedlist.tpp`.
   Hinweis: Wie in der Vorlesung besprochen muss die Implementierung von Template-Klassen im Header zu finden sein, da der Compiler sonst nicht wissen kann, für welche konkreten Typen er die Templates instantiieren muss. Alternativ können Sie ihre Implementierung auch in einer weiteren Datei `.tpp` realisieren und diese am Ende der Headerdatei `.hpp` einbinden (`include`).

3. Implementieren Sie eine Funktion `template<class T> void printList(const LinkedList<T>& list)`, welche Listen als konstante Referenz entgegennimmt und alle enthaltenen Index/Wert-Paare auf dem Bildschirm ausgibt. Diese Funktion soll die Debug-Ausgaben im gleich zu implementierenden Testprogramm übernehmen.

4. Schreiben Sie ein Testprogramm `test_linkedlist.cpp`, welches ein `List`-Objekt mit `double`-Elementen anlegt.

   - Legen Sie fünf Elemente an und weisen Sie jedem Element `i` den Wert `10*i` zu
   - Rufen Sie `printList()` auf, um die Liste vollständig auszugeben
   - Entfernen Sie anschließend das erste Element und fügen hinten ein Element mit dem Wert 100.0 an
   - Rufen Sie `printList()` erneut auf, um die Liste vollständig auszugeben
   - Versuchen Sie nun am Ende, ein Element anzusprechen, dessen Position nicht existiert. Überprüfen Sie, was zurückgegeben wird und stellen Sie sicher, dass auf dem Bildschirm eine Fehlermeldung erscheint.

5. Ihr Programm wird wahrscheinlich nicht sofort kompilieren. Woran liegt das, und wie können Sie das Problem beheben?

Hinweis: Überlegen Sie, ob Sie Ihre Implementierung der Liste mit mehr Funktionalität erweitern möchten. Überlegen Sie ausserdem gut, ob weitere Methoden Ihre Arbeit erleichtern könnten!

6 Aufgaben für Tag 4

6.1 Vererbung

6.1.1 Einführung

Mit der folgenden Übungsaufgabe sollen Sie die Vererbung anhand des vorigen `Car` Beispiels kennen lernen. Dazu sollen zwei neu zu erzeugenden Klassen von von `Car` erben:

- `CityCar`
  - neue Eigenschaft: aktuelles Ladegewicht, maximales Ladegewicht (denken Sie daran, das maximale Ladegewicht im Konstruktor zu erwarten)
  - neue Methoden: Beladen und entladen (beide sollen als Parameter ein Gewicht erwarten)
- `SportsCar`
  - neue Eigenschaft: Zustand des Verdecks
  - neue Methoden: Verdeck öffnen oder schließen

Vergessen Sie außerdem nicht, dass auch die Eigenschaften der Basisklasse `Car` initialisiert werden müssen, also müssen die von Ihnen zu implementierenden Konstruktoren von `CityCar` und `SportsCar` die entsprechenden Parameter ebenfalls erwarten, um diese an den Konstruktor von `Car` weitergeben zu können.

6.1.2 Testablauf

Folgende Testfälle sollen in Ihrem Testprogramm abgearbeitet werden:

1. `CityCar`
   (a) `CityCar`-Objekt mit 50 kg maximalem Ladegewicht
   (b) `CityCar`-Objekt beladen mit 25 kg
   (c) 25 kg wieder vom `CityCar`-Objekt entladen
2. `SportsCar`
   (a) `SportsCar`-Objekt erstellen
   (b) Verdeck öffnen
   (c) Verdeck schließen
   (d) `SportsCar`-Objekt ggf. wieder freigeben (wenn Sie das Objekt auf dem Heap erstellt hatten)

1. Erweitern Sie Ihr UML-Diagramm um die Klassen `CityCar` und `SportsCar`.
2. Implementieren Sie die Klassen in die Dateien `citycar.hpp`, `citycar.cpp`, `sportscar.hpp` und `sportscar.cpp`. Den Konstruktor der Basisklasse `Car` müssen Sie in der Initialisierungsliste der Konstruktoren von `CityCar` und `SportsCar` aufrufen.
3. Erzeugen Sie ein neues Testprogramm `cartest3.cpp`, das beide Klassen instantiiert und testet.
4. Führen Sie das Testprogramm aus, bewerten Sie das Ergebnis und korrigieren Sie eventuell auftretende Fehler.

6.2 Polymorphie

Auch wenn jedes Auto eine Hupe besitzt, haben Sie sicherlich schon einmal festgestellt, dass sich der Klang des honks von Typ zu Typ unterscheidet. Dieses Verhalten soll nun in einer Übungsaufgabe zur Polymorphie verbildlicht werden. Dazu haben Sie die Aufgabe, für die Klassen `Car`, `CityCar` und `SportsCar` jeweils eine virtuelle (und konstante) Funktion namens `honk` zu implementieren, welche unterschiedliche Ausgaben erzeugt:

- `Car::honk` soll „Beep“ ausgeben.
- `CityCar::honk` soll „TOOT!“ ausgeben.
- `SportsCar::honk` soll „Get off my lane!!“ ausgeben.

1. Erweitern Sie die drei oben genannten Klassen um die Methode `honk` – zunächst noch ohne das Schlüsselwort `virtual`.
2. Erzeugen Sie ein Testprogramm `cartest4.cpp` und implementieren Sie folgende Abläufe:
   (a) Erstellen Sie jeweils eine Instanz der Klassen `Car`, `CityCar` und `SportsCar`.
   (b) Rufen Sie die Methode `honk` der drei Instanzen auf.
   (c) Erstellen Sie einen Zeiger `Car *carPtr`, der zunächst auf Ihre `Car`-Instanz zeigt.
   (d) Rufen Sie `carPtr->honk()` auf.
   (e) Lassen Sie `carPtr` nun auf die `CityCar`-Instanz zeigen und rufen Sie `carPtr->honk()` nochmals auf.
   (f) Lassen Sie `carPtr` nun auf die `SportsCar`-Instanz zeigen und rufen Sie `carPtr->honk()` erneut auf.
3. Führen Sie das Testprogramm aus. Was wird ausgegeben?
4. Deklarieren Sie nun `honk` in den drei Headerdateien mit Hilfe des Schlüsselworts `virtual` als polymorphe Methoden.
5. Kompilieren Sie das Testprogramm erneut und führen Sie es aus. Was wird jetzt ausgeben? Können Sie eine Änderung feststellen?

7 Aufgaben für Tag 5

7.1 Iteratoren

Nachdem Sie in der Vorlesung Iteratoren kennen gelernt haben sollen Sie nun Ihrer Listenimplementierung einen Bidirektionaliterator hinzufügen. Fügen Sie dazu auch die Methoden `begin` und `end` hinzu.

1. Fügen Sie die Deklaration und Implementation für eine Iteratorklasse hinzu.
2. Achten Sie darauf, dass Ihr Iterator alle erforderten Operatoren unterstützt.
3. Testen Sie Ihren Iterator mit der Range Based For Loop und anderen Algorithmen der Standardbibliothek wie `std::find` und `std::for_each`.
4. Könnten Sie Ihren Iterator in einen Random Access Iterator erweitern? Wenn ja, welche Änderungen müssen Sie vornehmen? Wenn nein, wieso nicht?

Hinweis 1: Um einen Bidirektionaliterator implementieren zu können werden kleine Änderungen an Ihrer Liste vornehmen müssen!
Hinweis 2: Einige Algorithmen der Standardbibliothek erfordern, dass die Iteratorimplementierung mit einem „Tag“ zum Beispiel als `std::bidirectional_iterator_tag` gekennzeichnet ist. Dies ist am einfachsten lösbar, indem von `std::iterator` geerbt wird.

```
template<typename T>
class ListIterator : public std::iterator<std::bidirectional_iterator_tag, T>
```

7.2 Standard Template Library (STL)

Natürlich gibt es für die gängigen Containertypen bereits Implementierungen in der Standardbibliothek. In dieser Aufgabe sollen die Verwendung und Unterschiede der STL-Containerformate `vector`, `list` und `map` erläutert werden.

7.2.1 Performance-Unterschiede zwischen `vector` und `list`

Diese Aufgabe soll Ihnen die Möglichkeit geben, ein Gefühl dafür zu entwickeln, wie schnell verschiedene Operationen bei den unterschiedlichen STL-Containern sind. Konkret soll die Geschwindigkeit von Einfügeoperationen bei `std::vector` und `std::list` verglichen werden.

Hinweis: Die aktuelle Zeit erhalten Sie mit:

```
1 std::chrono::steady_clock::now()
```

Da `chrono` eine ehere „abstrakte“ Zeitmessung implementiert können, und müssen, Sie Zeitunterschiede explizit in eine Repräsentation casten. Für Sekunden z.B.:

```
1 auto duration = std::chrono::duration_cast<std::chrono::seconds>(endTime - startTime);
```

`duration.count()` gibt Ihnen dann die vergangene Zeit in Sekunden.

1. Erstellen Sie eine Datei `conainerperformance.cpp` und messen Sie die Zeiten für folgende Operationen, jeweils auf einem `std::vector` und `std::list` Objekt.

   (a) Fügen Sie 100.000.000 `int`-Werte **hinten** ein. Als einzufügenden Wert können Sie aus Einfachheitsgründen die Zahl 0 verwenden.

   (b) Löschen Sie die eben eingefügten Objekte von **hinten**.

   (c) Fügen Sie 100.000 `int`-Werte **vorne** ein. Als einzufügenden Wert können Sie aus Einfachheitsgründen die Zahl 0 verwenden.

   (d) Löschen Sie die eben eingefügten Objekte von **vorne**.

   (e) Ändern Sie in zufälliger Reihenfolge alle 50.000 Einträge der Container. Vergrößern Sie dazu zunächst den Container entsprechend und verwenden Sie einen Vector, der die Indices `0` bis `49.999` in zufälliger Reihenfolge enthält. Um den Index-Vector zu erstellen können Sie `std::shuffle` verwenden. Als einzufügenden Wert können Sie aus Einfachheitsgründen die Zahl 0 verwenden.

Was fällt Ihnen auf?

7.2.2 Verwendung einer `map`

Erstellen Sie ein Programm `map.cpp`, welches beliebig viele Namen mit dazugehörigen Telefonnummern einliest. Benutzen Sie hierfür den Namen (als `std::string`) als *key value* und die Telefonnummer als *mapped value* einer `map`. Geben Sie anschließend die gesamte Telefonliste mit Hilfe von Iteratoren alphabetisch sortiert aus.

8 Aufgaben für Tag 6

8.1 Callbacks

Callbacks werden besonders in Verbindung mit der `STL` interessant, z.B. wenn Sie in einem Container ein Objekt finden wollen, das bestimmte Eigenschaften erfüllt (`std::find_if`) oder, klassisch, wenn Sie auf allen Objekten eines Containers dieselbe Operation ausführen möchten.

8.1.1 Timer

Eine weitere Einsatzmöglichkeit für Callbackfunktionen sind Timer. Die Idee dabei ist, dass ein Thread mit einer Timerfunktion gestartet wird und entweder nach einer bestimmten Wartezeit oder periodisch die Callbackfunktion aufruft.
Hinweis: Multithreading führt gerne und schnell zu Raceconditions, wenn mehrere Threads dieselbe Ressource benutzen. Auch in diesem Beispiel wird es zu Raceconditions auf der Konsole kommen, allerdings soll Multithreading nicht tiefer behandelt werden und Raceconditions werden „in Kauf genommen". Wenn Sie dennoch interessiert sind, wie Sie ihre Threads gegen Raceconditions schützen können, bieten `std::mutex` eine einfache Lösung. Mehr zu Raceconditions finden Sie nach der Aufgabenbeschreibung hier 8.1.1.

1. Schreiben Sie eine Funktion `caller`, die drei Eingabeparameter annimmt, `std::chrono::seconds period`, `bool continuous` und `std::function<void()> callback`. `continuous` soll dabei angeben, ob `callback` endlos mit einer Periodendauer von `period` Sekunden, oder einmal nach `period` Sekunden aufgerufen werden soll.
   Hinweis: Überlegen Sie sich für den Fall, dass `caller` endlos ausgeführt werden soll, eine Möglichkeit die Schleife zu beenden wenn `main` beendet wird (**Hint:** `std::atomic`, siehe 8.1.1)! Die Möglichkeit den Thread zu beenden ist wichtig, um eventuelle Speicherlecks abfangen zu können!

2. Schreiben Sie drei Callable Objects, die jeweils unterschiedliche Ausgaben auf die Konsole schreiben. Möglichkeiten sind z.B. Zahlenfolgen wie Fibonacci, eine Karaoke-Box die, die nach und nach Lyrics ausgibt oder, ganz einfach, „Hello World". Die einzige Vorgabe hierbei soll sein, dass pro Aufruf nur eine Zeile ausgegeben wird, mindestens eines der Callable Objects nicht immer dieselbe Ausgabe liefert und alle Callable Objects unterschiedliche Typen sind (z.B. eine Funktion, ein Lambda und ein Functor).

3. `std::thread` bietet eine einfache Möglichkeit neue Threads zu starten und parallel auszuführen. Der Konstruktor eines `std::thread` Objektes nimmt dabei ein Callback Objekt und dazu passend die Parameter an. In dem Fall der `caller` Funktion könnte das also so aussehen:

   ```
   std::thread myThread(caller, 10s, false, foo);
   ```

   Starten Sie drei Threads die `caller` ausführen und jeweils eines der oben definierten Callable Objects als Callbackfunktion aufrufen. Überlegen Sie, welcher der Threads „endlos" laufen soll, also für welchen Thread `continuous == true` gelten soll.

Hinweis: Generell sollten alle gestarteten Threads irgendwann wieder zum Hauptthread `join`en. `myThread.join()` wartet dabei, bis `myThread` returned. Threads, die endlos laufen können mit `myThread.detach()` vom Programm gelöst werden, was allerdings u.U. zu Speicherlecks führen kann. Da in dieser Aufgabe kein Thread wirklich endlos läuft sondern spätestens mit beenden von `main` mit Hilfe Ihrer Überlegung aus 1 beendet werden soll, sollten Sie **alle** Ihre

Threads mit `myThread.join()` zum Ende von `main` auffangen.

Raceconditions Multithreading und Wenn mehrere Threads auf dieselbe Ressource zugreifen kommt es zwangsläufig zu Raceconditions. Im Grunde heißt das, dass zwei Threads/Prozesse gleichzeitig z.B. den Zustand einer Variable ändern wollen oder, wie in dieser Aufgabe, auf `stdout` schreiben wollen. Glücklicherweise ist Multithreading in `C++` kein Hexenwerk und es ist relativ einfach sich mit `std::atomic` und/oder `std::mutex` Variablen vor Raceconditions zu schützen.

- `std::atomic` kann hauptsächlich für kleine Datentypen genutzt werden und ist damit hauptsächlich für Counter o.Ä. interessant. `std::atomic` garantiert, dass Zuweisungen `operator=`, die prefix und postfix increment `++` und decrement `--` Operatore sowie Addition `operator+=`, Substraction `operator-=` und die binären Operationen `operator&=`, `operator|=` und `operator^=` atomar, das heißt ohne Unterbrechung durch andere Threads, geschehen.
- `std::mutex` wird genutzt um größere Strukturen oder Programmabschnitte zu schützen, die nicht mit atomaren Typen gesichert werden können. Dabei wird mindestens jeder Schreibzugriff auf die Ressource vorher mit dem Mutex ge-`lock`ed und nach dem Zugriff wieder ge-`unlock`ed. Zu beachten ist dabei, dass `lock` den Thread blockiert, also wartet bis der betreffende Mutex wieder `unlock`ed wurde. Wenn es also passieren kann, dass eine Ressource für einen längeren Zeitraum geblockt ist und die Ressource nicht unbedingt für den korrekten Ablauf des Programms nötig ist, kann auch `try_lock` genutzt werden, um nicht unnötig den Programmfluss zu blockieren.
  Beispiel für `std::cout`:

```
std::mutex outputMutex;
/* ... */
outputMutex.lock(); // waits until outputMutex is released and locks it
std::cout << "Hello World" << std::endl;
outputMutex.unlock(); // releases outputMutex
```

  Achtung: Bei Mutexen gehört zu jedem `lock` ein `unlock`. Ansonsten läuft man Gefahr Prozesse die auf einen Mutex warten für immer zu blockieren!

9 Aufgaben für Tag 7

9.1 Exceptions

Bei Auftreten eines Fehlers zur Laufzeit eines Programmes ist oft nicht unmittelbar klar, wie der Programmfluss fortgesetzt werden soll. Aus diesem Grund ist eine Art Signalisierung nötig, die aufgetretene Fehler an die Stelle meldet, bei der eine Entscheidung getroffen werden kann.

`C++` bietet mit Ausnahmen (Exceptions) einen Mechanismus zur Fehlersignalisierung an. Dazu wird unmittelbar beim Auftreten eines Fehlers mittels des `throw`-Schlsüsselworts eine Ausnahme geworfen. Geworfene Ausnahmen können schließlich mittels `try-catch`-Block abgefangen werden.

Das Werfen einer Ausnahme ist begleitet mit der Übergabe einer Variablen oder eines Objektes, das weitere Informationen über den Fehler enthält. Im einfachsten Fall kann z.B. eine Zahl geworfen werden.

```
1 double sqrt(const double number) {
2     result = 0.0;
3     if(number < 0.0) {
4         /* throw 1 */
5         throw static_cast<uint8_t>(1);
6     }
7     /* Implement sqrt function */
8     return result;
9 }
```

```
1 try {
2     double result = sqrt(-7.5);
3
4 } catch(uint8_t errorCode) {
5     std::cerr << "Caught␣Exception␣with␣errorCode␣"
6               << errorCode << std::endl;
7 }
```

Statt der Basisdatentypen können auch Klassenobjekte geworfen werden. Durch deren Verwendung kann man Ausnahmearten hierarchisch ordnen. In der `C++`-Standardbibliothek gibt es beispielsweise die Klasse `exception`, die als Basisklasse für alle von der Standardbibliothek geworfenen Ausnahmen dient.

9.1.1 Die `exception`-Klassenhierarchie

Abbildung 2 zeigt die `exception`-Klassenhierarchie. Durch das Abfangen von Ausnahmen des Typs `exception` sind automatisch auch die erbenden Klassen abgedeckt, d.h. so kann man bestimmen, von welcher Hierarchieebene man Ausnahmen abfangen möchte.

Als Aufgabe sollen Sie das Werfen einer Ausnahme in `List::at` implementieren. Dazu sinnvoll ist eine nähere Betrachtung der `out_of_range`-Klasse, die später auch geworfen werden soll. `out_of_range` ist in der Headerdatei `stdexcept` deklariert:

```
1 class out_of_range : public logic_error {
2 public:
3     out_of_range(const string& __arg);
4 };
```

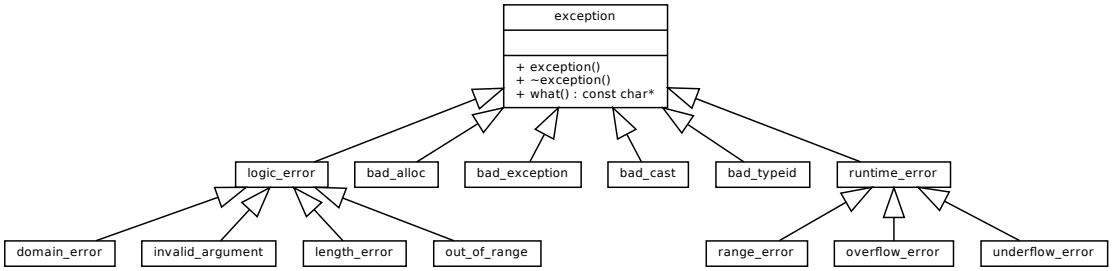

Abbildung 2: Die `exception`-Klassenhierarchie

Dem Konstruktor wird eine Referenz auf eine konstante Zeichenkette übergeben, die weitere Informationen über den Fehler in Textform enthält.

9.1.2 Aufgabe

1. Erweitern Sie in Ihrer List Implementierung die Methode `List::at` so, dass bei Überschreiten des maximalen Index ein `out_of_range` geworfen wird. Überlegen Sie sich eine sinnvolle Fehlermeldung, die an den Konstruktor von `out_of_range` übergeben werden kann.

2. Schreiben Sie ein Testprogramm `test_exceptions.cpp`, welches `out_of_range`-Ausnahmen abfängt und die übergebene Fehlermeldung auf dem Standard-Error-Kanal (`std::cerr`) ausgibt.
   Hinweis: Die Fehlermeldung, die an `out_of_range` übergeben wurde, kann mit der Methode `what()` ausgelesen werden.

10 Aufgaben für Tag 8

10.1 Qt

Der letzte Aufgabenblock soll Sie mit der Umgebung vertraut machen, welche Sie auch für den Projektteil verwenden werden.

Hier geht es um das Kennenlernen der Klassenbibliothek Qt, welche in ihren Anfängen als reine Grafikbibliothek gedacht war, im Laufe der Entwicklung aber deutlich umfangreicher geworden ist.

An dieser Stelle sollen Sie Qt zunächst dazu verwenden, um mit der Programmierung einer grafischen Benutzeroberfläche vertraut zu werden. Im zweiten Teil werden Sie ein einfaches Chatprogramm implementieren.

Um Ihnen den Einstieg in die grafische Programmierung zu erleichtern, werden wir zunächst eine einfache Oberfläche anlegen, die die grundlegenden Aufgaben eines Chats erfüllen soll.

10.1.1 Grafische Oberfläche

Diese sehr einfache Anwendung soll lediglich Nachrichten einlesen und weiter verschicken können, also die Funktion eines Echos erfüllen. Zur Kontrolle der richtigen Funktionalität werden wir die Nachrichten nur an uns selbst schicken. Eine Netzwerkfunktion wird deshalb vorerst nicht benötigt. Das Echo-Programm soll folgende grafische Bedienelemente bereitstellen:

- ein Texteingabefeld, welches Ihren Namen (Nickname) enthält
- ein Texteingabefeld, in das die zu sendende Nachricht eingetragen werden soll
- ein Textfeld, das das Gespräch beinhalten soll
- einen „Senden“-Knopf, der die Übertragung der im Texteingabefeld eingegebenen Nachricht auslösen soll

Es gibt viele Grafik-Elemente, die dem Qt-Designer zur Verfügung stehen. Nehmen Sie sich also Zeit zu überlegen, welche Objekte die geforderten Rollen am besten erfüllen. Weiterhin können schon beim Anlegen der Elemente verschiedene Funktionen oder Regeln - wie das Verbieten von direkten Eingaben - im Designer festgelegt werden.

1. Erstellen Sie im Qt Creator ein Projekt namens `Echo` als „Qt Gui Application“. Bei den Details zu den Klasseninformationen tragen Sie als Basisklasse `QWidget` und als UI-Datei `echogui.ui` ein. Ändern Sie den Klassennamen auf `EchoApp`, nennen Sie die Headerdatei `echoapp.h` und Quelldatei `echoapp.cpp`.

2. Wechseln Sie nach der Konfiguration Ihres Projektes auf den Design-Tab, indem Sie links einen Doppelklick auf den Eintrag `echogui.ui` ausführen. Hier können Sie nun mittels Drag'n'Drop die oben beschriebenen Anforderungen für die grafische Oberfläche erfüllen.

3. Wechseln Sie nun zurück auf den Quellcode-Editor und fügen Sie der `EchoApp`-Klasse einen Slot namens `send()` hinzu. Dieser soll den im Nachrichtenfeld eingegebenen Text in das Gesprächsfeld übernehmen.

4. Erweitern Sie den Konstruktor der `EchoApp`-Klasse nun insoweit, dass das Signal `clicked()` des „Senden“-Buttons mit dem zuvor implementierten `send()`-Slot verbunden wird.

11 Aufgaben für Tag 9

11.1 Chat

Nach dem kurzen Einstieg liegt die Erweiterung des schon erstellten Programms zu einem grafischen Chat-Programm nahe, welches Chats zwischen mehreren Personen ermöglichen soll. In der Endausbaustufe sollen Sie einen Chat-Client implementieren, welcher per Netzwerk mit einem gegebenen Chat-Server kommuniziert.

Aufbauend auf das vorhandene Echo-Programm soll der Chat-Client folgende grafische Bedienelemente bereitstellen:

- für den Chat
  - ein einzeiliges Texteingabefeld, welches Ihren Namen (Nickname) enthält
  - ein einzeiliges Texteingabefeld, in das die zu sendende Nachricht eingetragen werden soll
  - ein mehrzeiliges Textfeld, das eine Historie der gesendeten und empfangenen Nachrichten anzeigen soll
  - ein „Senden“-Button, der die Übertragung der im Texteingabefeld eingegebenen Nachricht auslösen soll (alternativ kann die Übertragung auch ausgelöst werden durch das Betätigen der Enter-Taste im Texteingabefeld der Nachricht)
- für die Netzwerkkommunikation
  - ein einzeiliges Texteingabefeld für die „Server-Adresse“.
  - ein einzeiliges Texteingabefeld für den „Server-Port“, in dem eine Zahl zwischen 1024 und 65535 eingegeben werden soll. Die genaue Bedeutung dieses Feldes wird später erklärt.
  - ein „Verbinden“-Button, der die Verbindung zum angegebenen Chat-Server startet
  - ein „Trennen“-Button, der eine offene Verbindung zu einem Chat-Server trennt

11.2 Netzwerkprogrammierung

Dieser Abschnitt soll Ihnen einen Überblick über die hier wichtigen *Grundlagen der Netzwerkkommunikation* geben, sowie ein *Protokoll* festlegen, über welches sich Chat-Server und Client miteinander unterhalten können. Hier kann Ihnen jedoch nur ein oberflächlicher Überblick über Netzwerktechnik gegeben werden. Für ein tiefergehendes Verständnis sei auf die vielfältig vorhandene Fachliteratur verwiesen.

Grundsätzlich hat jeder am Internet angeschlossene Rechner eine eindeutig identifizierbare Adresse, die so genannte *IP-Adresse*. Übertragene Daten sind mit IP-Adresse des Absenders und des Empfängers gekennzeichnet.

Im Internet werden Daten *paketorientiert* übertragen. Dazu werden Daten in Pakete „zerstückelt“, die alle einzeln und prinzipiell unabhängig voneinander übertragen werden. Jedes Paket enthält die Absender- und die Zieladresse.

Im Zusammenhang mit dieser Zerstückelung kann es passieren, dass einzelne Pakete verloren gehen oder sich die Reihenfolge der Pakete ändert, sodass sich der gesendete Datenstrom auf der Empfängerseite nicht immer vollständig rekonstruieren lässt. In vielen Fällen ist dies aber eine zwingende Voraussetzung für eine störungsfreie Kommunikation.

Das so genannte *Transmission Control Protocol (TCP)* setzt eine solche störungsfreie Kommunikation um, indem beispielsweise Fehlererkennungsverfahren eingesetzt werden. Außerdem

werden verloren gegangene oder in der falschen Reihenfolge ankommende Pakete durch eine Nummerierung erkannt und gegebenenfalls neu von der Gegenseite angefordert. TCP arbeitet *verbindungsorientiert*, d.h. als Programmierer kann man annehmen, dass mit TCP eine **feste Verbindung** zum Kommunikationspartner besteht, obwohl IP die Daten in zunächst nicht zusammenhängenden Paketen überträgt.

Um eine TCP-Verbindung aufzubauen, sind folgende Schritte nötig:

1. Zunächst wird ein Server benötigt, der auf einem so genannten *Port* (vergleichbar mit einer Durchwahl) auf eingehende Verbindungen wartet („listened“).

2. Ein Client kann durch Angabe der Adresse des Servers und des Server-Ports eine Verbindungsanfrage an den Server stellen.

3. Wenn der vom Client angefragte Server-Port geöffnet ist, d.h. auf eingehende Verbindungen wartet, und auch eine ggf. vorhandene Firewall die Verbindung erlaubt, so akzeptiert der Server die Verbindung.

4. Server und Client haben nun eine Verbindung, mittels derer Daten zuverlässig miteinander ausgetauscht werden können.

Dienste, die auf TCP basieren, sind allseits bekannt. Wichtige Vertreter sind HTTP („Web-Surfen“) sowie SMTP und IMAP (E-Mail).

Der gegebene Chat-Server erwartet folgendes Datenformat:

1. Nickname als `QString`-Objekt im **Big-Endian**-Format

2. Nachricht als `QString`-Objekt im **Big-Endian**-Format

Nutzen Sie für die Übertragung die von Qt bereitgestellten Klassen für die TCP-Schnittstelle (`QTcpSocket` und `QTcpServer`) sowie die Stream-Klassen (`QDataStream` und `QTextStream`). Entscheiden Sie selbst, welche Klassen Sie für Ihre Implementierung benötigen und benutzen wollen. Informieren Sie sich dazu in der Qt-Dokumentation `http://doc.qt.io/qt-4.8` und `http://doc.qt.io/qt-4.8/classes.html`.

Hinweis: In der Netzwerkprogrammierung muss der Programmierer normalerweise darauf achten, dass die richtige Anzahl an Bytes gesendet und empfangen werden. Dafür gibt es Funktionen, die es dem Programmierer ermöglichen die Anzahl gesendeter und zum Lesen verfügbarer Bytes von der Netzwerk-Schnittstelle zu erfahren. In der Praxis wird dazu Bytes einzeln gesendet und überprüft, ob sie korrekt gesendet wurde. Zusätzlich wird vor dem Senden der Nachricht die Länge der Nachricht (auch byteweise) an den Empfänger gesendet. Auf der Empfängerseite wir dann zuerst die Länge der Nachricht gelesen und gewartet, bis entsprechend viele Bytes zum Lesen bereit sind. Die Kommunikationspartner müssen sich dabei an ein Vorgeschriebenes Protokoll halten, um eine problemlose Kommunikation zu ermöglichen. In diesem Protokoll wird zum Beispiel der Datentyp der Nachrichtenlänge und die Codierung der Nachricht (UTF-8, UTF-16,...) festgelegt.

Wie oben beschrieben sollen Sie in Ihrer Implementierung die Qt-Klasse `QString`, die Klassen für die TCP-Schnittstelle (`QTcpSocket` und `QTcpServer`) sowie die Stream-Klassen benutzen. Hierbei wird von Qt bereits ein Protokoll Implementiert, in dem festgelegt wird, wie div. Qt-Klassen gesendet werden. Dabei wird ein `QString` als ein Paket aus seiner Länge (als `quint32`) und des Textes (als UTF-16) gesendet (siehe `http://doc.qt.io/qt-4.8/datastreamformat.html`). Für Sie heißt das, dass Sie die Länge des Textes nicht extra senden müssen und auch nicht überprüfen müssen, ob die Nachricht vollständig gesendet wurde. Diese Aufgabe wird von der

verwendeten Stream-Klasse übernommen. In Projekten, in denen **nicht** Qt oder keine Qt-Stream Klasse verwendet wird müssen Sie diese Probleme trotzdem berücksichtigen!

Hinweis: Die Netzwerk-Funktionen des Qt-Frameworks erwarten die zu übertragenden Daten – wie auch die meisten anderen Netzwerk-Funktionen – byte-weise. Daten, welche größer als ein Byte sind, müssen folglich zur Übertragung in mehrere Bytes zerlegt werden. Hierbei ergibt es sich nicht automatisch, in welcher Reihenfolge die Bytes übertragen werden. Es haben sich zwei „konkurrierende“ Ansätze ergeben.

- Übertragungen im sogenannten *Little-Endian*-Format bedeuten, dass das Byte mit der geringsten Wertigkeit als erstes übertragen wird. Die darauf folgenden Bytes werden mit aufsteigender Wertigkeit übertragen.
- Übertragungen im *Big-Endian*-Format bedeuten, dass als Erstes das Byte mit der höchsten Wertigkeit zu übertragen ist; danach sollen alle anderen Bytes in absteigender Reihenfolge der Wertigkeit folgen.

Als Beispiel soll die Übertragung der Zahl `uint32_t`-Zahl 0x12345678 herangezogen werden.

- Im Little-Endian-Format würde folgende Bytefolge übertragen:
  `0x78`, `0x56`, `0x34`, `0x12`
- Im Big-Endian-Format würde folgende Bytefolge übertragen:
  `0x12`, `0x34`, `0x56`, `0x78`

Weitere Beispiele und Erklärungen finden sich auch unter `http://en.wikipedia.org/wiki/Endianness`.

11.3 GUI

An dieser Stelle sollen Sie mit der Implementierung der grafischen Benutzeroberfläche beginnen; die Netzwerkfunktionalität ist hier noch nicht erforderlich, stattdessen sollen lediglich lokal eingegebene Chat-Nachricht in der Historie erscheinen. Sie werden hier entscheidende Parallelen zur vorherigen Aufgabe feststellen, sollten jedoch nichtsdestotrotz dies als Grundstock für folgende Aufgaben wiederholen.

Die Historie soll Nachrichten in folgender Form anzeigen:

```
<Nickname> Nachricht
```

1. Erstellen Sie im Qt Creator ein Projekt namens `chat` und speichern Sie die UI-Datei unter dem Namen `chatapp.ui` und die Klasse als `ChatApp`, welche von der Basisklasse `QWidget` erbt. Legen Sie die grafischen Bedienelemente im Qt Designer an.
2. Fügen Sie der `ChatApp`-Klasse einen Slot namens `send()` hinzu. Dieser soll zunächst den im Nachrichtenfeld eingegebenen Text in das Historienfeld übernehmen.
3. Erweitern Sie den Konstruktor der `ChatApp`-Klasse nun insoweit, dass das Signal `clicked()` des „Senden“-Buttons mit dem zuvor implementierten `send()`-Slot verbunden wird.

11.4 Chat-Funktionalität über das Netzwerk

Abbildung 3 zeigt den Zustandsautomaten der Netzwerkverbindung des Clients. Zur Implementierung der Netzwerkfunktionalitäten ist die Klasse `QTcpSocket` näher zu betrachten. Eine genaue Beschreibung dieser Klassen finden Sie unter `http://qt-project.org/doc/qt-4.8/`

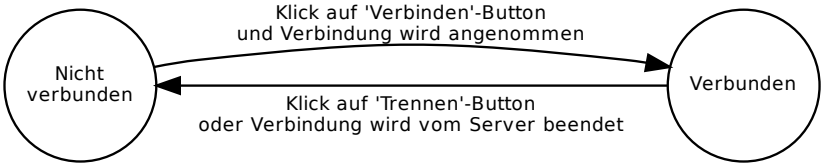

Abbildung 3: Zustandsautomat des Clients

`network-programming.html`.

1. Implementieren Sie einen Slot `connectDisconnect()` und verbinden Sie diesen mit dem Signal `clicked()` des "Verbinden/Trennen"-Buttons. Der Slot soll das oben dargestellte Zustandsdiagramm implementieren.
   Hinweis: Um die Netzwerkklassen von Qt zu benutzen muss in der Projekt-Datei (`*.pro`) die folgende Zeile hinzufügen:
   `QT += network`

2. Zustandsübergänge können sich durch Klicken des „Verbinden/Trennen"-Buttons, aber auch durch Verbindungsaufbau (nur Server) und -abbruch der Gegenseite (Server und Client) ergeben. Suchen Sie nach einem Signal von `QTcpSocket`, welches die Trennung der Verbindung signalisiert. Implementieren Sie unter Berücksichtigung der dargestellten Zustandsautomaten dazu passende Slots, und verbinden Sie Signals und Slots miteinander.
   Hinweis: Sollten Sie bei der Klasse `QTcpSocket` nicht fündig werden, so sehen Sie sich auch bei deren Basisklassen um, von denen Signale vererbt werden.

3. Nachdem Auf- und Abbau der Verbindung sauber implementiert sind, können Sie den zuvor erstellten Slot `send()` so erweitern, dass er den im Nachrichtenfeld eingegebenen Text über das Netzwerk an den Server überträgt. Verwenden Sie dazu die oben erwähnten Klassen für die TCP-Schnittstelle und die Stream-Klassen.

4. Nun muss auch das Empfangen von Daten abgearbeitet werden. Auch dazu bietet `QTcpSocket` entsprechende Signale. Finden Sie ein geeignetes Signal, welches die Ankunft neuer Nachrichten anzeigt. Implementieren Sie dazu einen Slot, der empfangene Nachrichten in folgendem Format in das Historien-Textfeld einträgt:
   `<NicknameAbsender> Nachricht`

5. Testen Sie Ihr Programm, indem Sie mit Ihren Kommilitonen chatten.
   Hinweis: Ihre im Netzwerk sichtbare IP-Adresse bekommen Sie mit dem Befehl `ifconfig eth0` heraus. Sie wird nach `inet addr` (IPv4) bzw. `inet6 addr` (IPv6) angezeigt.