

Spieledesign mit C++1x

Spieledesign für den informationstechnischen Unterricht an
der exemplarischen Applikation ncSoko++

Version 0.3, Januar 2020

Autor:
doraus@bbs-me.de



Inhaltsverzeichnis

Inhaltsverzeichnis	I
Abkürzungsverzeichnis	V
Abbildungsverzeichnis	VII
1. Einleitung	10
2. Hinweise zum Dokument	11
3. Erstellung und Bearbeitung eines C++-Programms	12
3.1 Das erste C++-Programm	12
3.2 Erzeugen des ausführbaren Programms	13
3.3 Aufgabe des C++-Compilers	14
3.4 Der Editor vi für die Eingabe des C++-Quellcodes	14
3.5 C/C++-Style-Guidelines	15
4. Sokoban - Das Computerspiel	17
4.1 Spieleprinzip	17
4.2 Lagerhaus-Design	17
5. Design und Entwicklung des Computerspiels Sokoban	19
5.1 Grafische Ausgabe mit der Ncurses-Bibliothek	20
5.2 <u>Part 00</u> – Start mit Standard-Ein- und Ausgaben	20
5.3 <u>Part 01</u> – Start mit der Ncurses-Bibliothek	22
5.3.A Endlosschleife mit while, kopfgesteuert	22
5.3.B Hallo Ncurses, string, auto und range-based for	23
5.3.B.1 Der Datentyp string	23
5.3.B.2 Der Datentyp auto	24
5.3.B.3 Die Range-based for-Schleife	24
5.3.C Das Ncurses-Koordinatensystem	25
5.3.D Triggerpunkt Part 01	26
5.4 <u>Part 02</u> – Ein farbiges Ncurses-Fenster mit Eingaben	27
5.4.A Altes Programm, neuer Aufbau	27
5.4.A.1 Der Datentyp int	30
5.4.A.2 Der Datentyp bool	31
5.4.A.3 Funktionsdefinition und Funktionsaufruf	31

5.4.A.4	Die while-Schleife	32
5.4.A.4.1	Bedingungen in Verzweigungen und Schleifen.....	32
5.4.A.5	Verzweigungen mit switch-case	33
5.4.B	Farben und Boxen mit Ncurses	34
5.4.C	Das Resultat	35
5.4.D	Triggerpunkt Part 02	35
5.5	<u>Part 03</u> – Ein beliebiges Spielfeld	36
5.5.A	Exkurs – Objektorientierte Programmierung	36
5.5.B	Part 03-1 – Das Gameboard-Modul via UML	36
5.5.B.1	Exkurs: Das Klassendiagramm.....	37
5.5.B.1.1	Name der Klasse.....	37
5.5.B.1.2	Eigenschaften der Klasse.....	37
5.5.B.1.3	Methoden der Klasse	37
5.5.B.1.4	Sichtbarkeiten	38
5.5.B.1.5	Assoziationen zwischen Klassen	38
5.5.B.2	Die Implementierung.....	39
5.5.B.3	Triggerpunkt Part 03-1	42
5.5.C	Part 03-2 – Das Pixel als Zusammenschluss	42
5.5.C.1	Anlegen eines Point-Objekts	43
5.5.C.2	Schreiben der Point-Komponenten.....	43
5.5.C.3	Lesen der Point-Komponenten	43
5.5.D	Das Resultat	44
5.5.E	Triggerpunkt Part 03	44
5.6	<u>Part 04</u> – Das Sokoban-Spielfeld	45
5.6.A	Die Ncurses-Ausgabe der Aufgabe	46
5.6.B	Der Datentyp char	46
5.6.C	Die ASCII-Tabelle.....	47
5.6.D	Das Resultat	48
5.6.E	Die Zahlensysteme	48
5.6.E.1	Das binäre Zahlensystem.....	48
5.6.E.2	Das hexadezimale Zahlensystem.....	49
5.6.E.3	Umrechnung Bin in Hex und Hex in Bin	49

5.6.F	Triggerpunkt Part 04	50
5.7	<u>Part 05</u> – Ein Spieler kommt ins Spiel	51
5.7.A	Part 05-1 – Ein Spieler betritt das Spielfeld.....	51
5.7.A.1	Das Resultat.....	52
5.7.B	Part 05-2 – Der Spieler bewegt sich.....	52
5.7.B.1	Das Resultat.....	53
5.7.B.2	Die Spielerrestauration	53
5.7.C	Part 05-3 – Der Spieler und die Kollisionsabfrage	54
5.7.C.1	Das Resultat.....	55
5.7.C.2	Die if-Verzweigung	55
5.7.C.3	Der Debugger GDB und GDBTUI.....	55
5.7.C.4	Triggerpunkt Part 05.....	56
5.8	<u>Part 06</u> – Kisten und Ziele positionieren.....	57
5.8.A	Das Ergebnis	58
5.8.B	Triggerpunkt Part 06	58
5.9	<u>Part 07</u> – Die Kisten verschieben	59
5.9.A	Part 07-1 – Das Logger-Modul	59
5.9.B	Part 07-2 – Kiste für Kiste schieben	60
5.9.B.1	Zeiger-Variablen und ihre Funktion	61
5.9.B.2	Die Null-Zeiger-Konstante nullptr	62
5.9.C	Part 07-3 – Restauration der Ziele	63
5.9.C.1	Der Vektor-Container	64
5.9.D	Das Ergebnis	65
5.9.E	Triggerpunkt Part 07	65
5.10	<u>Part 08</u> – Aufgabe erfüllt?	66
5.10.A	Das Ergebnis.....	66
5.10.B	Triggerpunkt Part 08.....	66
5.11	<u>Part 09</u> – Aufgabe laden bitte schön.....	68
5.11.A	Erstellen einer neuen Aufgabe – Das Level-Design.....	68
5.11.B	Part 09-1 – Lesen einer neuen Aufgabe in ein Vektor	69
5.11.B.1	Textdateien-Schnittstelle	70
5.11.B.2	Maximale Anzahl Spalten und Zeilen bestimmen.....	70

5.11.C	Part 09-2 – Zuordnen der Vektor-Elemente.....	71
5.11.D	Part 09-3 – Datei als Kommandozeilenparameter.....	72
5.11.D.1	Die Kommandozeilenparameter.....	72
5.11.E	Das Ergebnis.....	74
5.11.F	Triggerpunkt Part 09.....	74
6.	Erweiterung des Computerspiels Sokoban.....	75
6.1	<u>Part 10</u> – Moves und Pushes.....	75
6.1.A	Part 10-1 – Ein zusätzliches Fenster für die Anzeige.....	75
6.1.A.1	Das Ergebnis.....	76
6.1.B	Part 10-2 – Anzeige mit Leben füllen.....	76
6.1.B.1	Das Ergebnis.....	77
6.1.C	Triggerpunkt Part 10.....	77
6.2	<u>Part 11</u> – Eine Ampel als Abwärtszähler.....	78
6.2.A	Part 11-1 – Die Timer-Threads.....	78
6.2.A.1	Ergebnis.....	79
6.2.B	Part 11-2 – Die Ampel-Timer.....	80
6.2.B.1	Ergebnis.....	81
6.2.C	Part 11-3 – Wartezeit per Kommandozeile.....	82
6.2.D	Triggerpunkt Part 11.....	82
6.3	<u>Part 12</u> – Spieler, Kisten und Ziele einfärben.....	83
6.3.A.1	Ergebnis.....	83
6.3.A.2	Triggerpunkt Part 12.....	83
7.	Zusatz – Erweiterungen.....	84
8.	Anlagen.....	86
8.1	ASCII-Tabelle.....	86
8.2	C++ Referenzkarte.....	87
8.3	GDB Referenzkarte.....	89

Abkürzungsverzeichnis

a.a.O.	am angegebenen Ort
Abs.	Absatz
ASCII	American Standard Code for Information Interchange
Aufl.	Auflage
Bd.	Band
Bsp.	Beispiel
bspw.	beispielsweise
bzw.	beziehungsweise
ders.	Derselbe
DBMS	Database Management System
dies.	dieselbe(n)
Diss.	Dissertation
d. h.	das heißt
Dok.	Dokument
et al.	und andere
etc.	und so weiter
f.	(die) folgende
ff.	(die) folgenden
Fort.	Fortsetzung
ggf.	gegebenenfalls
hj.	halbjährlich
Hrsg.	Herausgeber
hrsg. v.	herausgegeben von
i.d.R.	in der Regel
Jg.	Jahrgang
jhrl.	jährlich
mtl.	monatlich
N.F.	Neue Folge
Nr.	Nummer

o.Ä.	oder Ähnliche(s)
o.ä.	oder ähnlich
o.J.	ohne Jahresangabe
o.O.	ohne Ortsangabe
o.V.	ohne Verfasserangabe
S.	Seite
s.	siehe
Sp.	Spalte
s.o.	siehe oben
sog.	so genannte
SuS	Schülerinnen und Schüler
Tz.	Textziffer
u.	und
u.a.	und andere
u.Ä.	und Ähnliche(s)
u.ä.	und ähnlich
Verf.	Verfasserin / Verfasser
Verl.	Verlag
vgl.	vergleiche
vj.	vierteljährlich
z.B.	zum Beispiel
zit. nach	zitiert nach
TUI	Text User Interface

Abbildungsverzeichnis

Abbildung 1: Das Hello World-Programm im Quellcode C++.....	12
Abbildung 2: Kompilierungsvorgang, vom Quellcode zum ausführbaren Programm....	13
Abbildung 3: detaillierter Kompilierungsvorgang unseres "Hello World"-Programms .	14
Abbildung 4: Der vi in der Konsole	15
Abbildung 5: Walkthrough vimtutor	15
Abbildung 6: Aufgaben-Format	18
Abbildung 7: Das Konsole-Spiel Sokoban.....	19
Abbildung 8: Eine Java-Swing-Umsetzung des Spiels Sokoban	20
Abbildung 9: Koordinatensystem in Neurses.....	25
Abbildung 10: Tastaturbelegung für die Bewegung des Spielers	27
Abbildung 11: Das farbige Fenster als Spielfläche	35
Abbildung 12: Klassendiagramm Part 03-1	36
Abbildung 13: UML-Komposition.....	38
Abbildung 14: erweitertes Klassendiagramm Part 03-2.....	43
Abbildung 15: Das Spielfeld 10x10 Pixel.....	44
Abbildung 16: erweitertes Klassendiagramm Part 04.....	45
Abbildung 17: Testaufgabe mit Spielfeldumrandung.....	45
Abbildung 18: Ausschnitt der ASCII-Tabelle mit dem #-Zeichen	47
Abbildung 19: Das Spielfeld	48
Abbildung 20: Testaufgabe mit Spieler	51
Abbildung 21: erweitertes Klassendiagramm Part 05-1.....	51
Abbildung 22: Der Spieler auf dem Spielfeld.....	52
Abbildung 23: erweitertes Klassendiagramm Part 05-2.....	52
Abbildung 24: Der Spieler bewegt sich (aber noch ohne Kollisionsabfrage).....	53
Abbildung 25: Restauration von Inhalten bei Bewegungen.....	54
Abbildung 26: Der Spieler bewegt sich (mit Kollisionsabfrage).....	55
Abbildung 27: Komplette Testaufgabe mit Kisten und Ziele.....	57
Abbildung 28: erweitertes Klassendiagramm Part 06.....	57
Abbildung 29: Spieler läuft über Kisten mit Restauration	58
Abbildung 30: erweitertes Klassendiagramm Part 07-1.....	59
Abbildung 31: Verschieben einer Kiste	60
Abbildung 32: erweitertes Klassendiagramm Part 07-2.....	61
Abbildung 33: Objekte und Zeiger auf diese Objekte im Speicher	62
Abbildung 34: erweitertes Klassendiagramm Part 07-3.....	63
Abbildung 35: Spieler schiebt Kisten auf Ziele inkl. Restauration.....	65
Abbildung 36: erweitertes Klassendiagramm Part 08.....	66
Abbildung 37: Beispiel-Aufgabe mit dem vi erstellen.....	68
Abbildung 38: erweitertes Klassendiagramm Part 09-3.....	72
Abbildung 39: Kommandozeilenargumente im Speicher	73
Abbildung 40: Aufgabe ist übergeben worden und kann gespielt werden.....	74
Abbildung 41: erweitertes Klassendiagramm Part 10-1.....	75
Abbildung 42: Das neues Fenster für die Anzeige von Statusanzeigen	76
Abbildung 43: erweitertes Klassendiagramm Part 10-2.....	76
Abbildung 44: Das Statusfenster mit den Moves und Pushes.....	77
Abbildung 45: erweitertes Klassendiagramm Part 11-1.....	78
Abbildung 46: erweitertes Klassendiagramm Part 11-2.....	80

Abbildung 47: Die Status-Ampel in Aktion	81
Abbildung 48: erweitertes Klassendiagramm Part 11-3.....	82
Abbildung 49: Spieler, Kisten und Ziele eingefärbt.....	83
Abbildung 50: ASCII-Code Tabelle	86
Abbildung 51: C++ Referenzkarte	88
Abbildung 52: GDB Referenzkarte.....	90

*“Nie haben die Menschen mehr Geist bewiesen, als bei den Spielen,
die sie erfunden haben. Ganz allgemein gesprochen
sind es immer die geistreichsten, die die Spiele erfinden und
die Dümmersten, die diese Spiele am besten spielen.”*

Gottfried Wilhelm Leibnitz (1646 -1716)

*Die Weltgesundheitsorganisation (WHO) hat auf ihrer Jahresversammlung 2019 in Genf endgültig die elfte Version der Internationalen Klassifikation der Krankheiten (ICD-11) verabschiedet und damit auch **Computerspielesucht als Krankheit** anerkannt. Eigentlich handelt es sich dabei um eine weitere Formalie: **Gaming Disorder** wurde bereits 2018 nach langen Diskussionen in den Katalog der Krankheitsbilder aufgenommen. Damit könne künftig Behandlungskosten von den Krankenkassen übernommen werden.*

<http://www.golem.de>, 24.05.2019, Peter Steinlechner

1. Einleitung

Computerspiele haben in der Informatik einen festen Stellenwert. Seit den 1980er Jahren haben sie ihren Siegeszug angetreten. Die ersten Spiele waren textbasiert und die Präsentation solcher Spiele war besonders schlicht und einfach gehalten. Bei diesen Spielen stand die Idee – also der Sinn des Spiels – im Vordergrund. Sie hatten eine große Fan-Gemeinde, weil sie modifizierbar waren oder weil Maps oder Levels dazu gebaut werden konnten. Diese Spiele übten deswegen eine große Anziehungskraft auf überwiegend junge Menschen aus und wurden über einen langen Zeitraum gespielt. Mit der Zeit wurden Spiele immer komplexer und es entwickelte sich ein Industriezweig daraus. Spiele wurden vermarktet und sie bilden Lebensgrundlage vieler Spieleentwickler.

Aus technischer Sicht ist ein Computerspiel hoch komplex und stellt hohe Ansprüche an das Knowhow eines Entwicklers. In einem modernen Spiel sind viele Aspekte der Informatik vereint:

- Hardwarenahe Programmierung einer Grafikengine ggf. in Assembler
- Ein- und Ausgaben zwischen Programm und Spieler (MMI)
- Realistische Simulation von statischen Landschaften in 3D
- Realistische Simulation von bewegten Objekten in 3D (Motion Capturing)
- Plattformunabhängigkeit (Windows/Linux/Unix/Mac-OS/IOS/Android)

Die Entwicklung eines eigenen Computerspiels dient einem tiefgreifenden Verständnis von Programmier- und Designtechniken in der Informatik.

2. Hinweise zum Dokument

In hellblauen Kästen werden Quellcodes dargestellt!

In dunkelblauen Kästen werden die Ausgaben oder Programmaufrufe dargestellt!

In gelben Kästen werden besondere Informationen dargestellt!

3. Erstellung und Bearbeitung eines C++-Programms

Ok, ein Computerprogramm benötigt ein Startpunkt, der es dem Betriebssystem erlaubt, das Programm auszuführen. In allen C++ -Anwendungen muss deshalb eine Funktion `main` existieren, also auch unser Computerspiel.

3.1 Das erste C++-Programm

Die erste Applikation ist das berühmte `Hello World`-Programm, das folgendermaßen aufgebaut sein muss. Es ist in einer Datei `hello.cpp` gespeichert.

```

1 /**
2  * Hello World...
3  */
4
5 #include <iostream>
6
7 int main()
8 {
9     std::cout << "Hello World" << std::endl;
10    return 0;
11 }
```

Abbildung 1: Das Hello World-Programm im Quellcode C++

Die untenstehende Tabelle gibt Aufschluss über die Funktion der einzelnen Anweisungen.

Zeile	Funktion
1-3	Ein Kommentar wird immer mit <code>/*</code> eingeleitet und mit <code>*/</code> beendet. Alles dazwischen ist Kommentar und wird von Compiler nicht beachtet.
5	<code>iostream</code> stellt die Verknüpfung zur <code>iostream</code> -Bibliothek her. Das wird mit der Präprozessoranweisung <code>#include</code> durchgeführt.
7	Die Funktion <code>int main()</code> ist für den Einsprung des Betriebssystems zwingend erforderlich. <code>int</code> ist der Datentyp des Rückgabewerts (<code>return 0</code>).
8,11	Die beiden Klammern <code>{</code> und <code>}</code> signalisieren einen Funktionsblock. Innerhalb der Klammern muss der Code stehen, der bei Aufruf der Funktion (<code>main</code>) ausgeführt werden soll.
9	<code>std::cout</code> ist eine Bibliotheksmethode, die Variablen oder Strings auf dem Bildschirm ausgibt. Getrennt werden die Variablen oder Strings mit <code><<</code> . Am Ende der <code>cout</code> -Zeile kann die Anweisung <code>std::endl</code> stehen.

	Die sorgt dafür, dass die nächste Ausgabe in einer neuen Zeile angefangen wird.
10	<code>return</code> gibt einen Wert (hier 0) an das Betriebssystem zurück. Damit kann der Entwickler dem Betriebssystem mitteilen, ob sein Programm ordnungsgemäß oder mit einem Fehler beendet wurde.

3.2 Erzeugen des ausführbaren Programms

Unser `Hello World`-Programm ist in C++-Quellcode geschrieben. Die Funktion des Programms ist für alle Entwickler, die den C++-Dialekt beherrschen, verständlich und nachvollziehbar. Der Mikroprozessor oder Mikrocontroller ist nicht in der Lage, den Quellcode zu verstehen, um ihn dann auszuführen. Er benötigt einen weitaus schwieriger nachzuvollziehenden Dialekt, den **Binärcode**. Der Inhalt des Binärcodes (Zielprogramm) besteht nur aus zwei unterschiedlichen Zahlen 0 und 1, daher der Name (binär, lat. "aus zwei Einheiten bestehend").

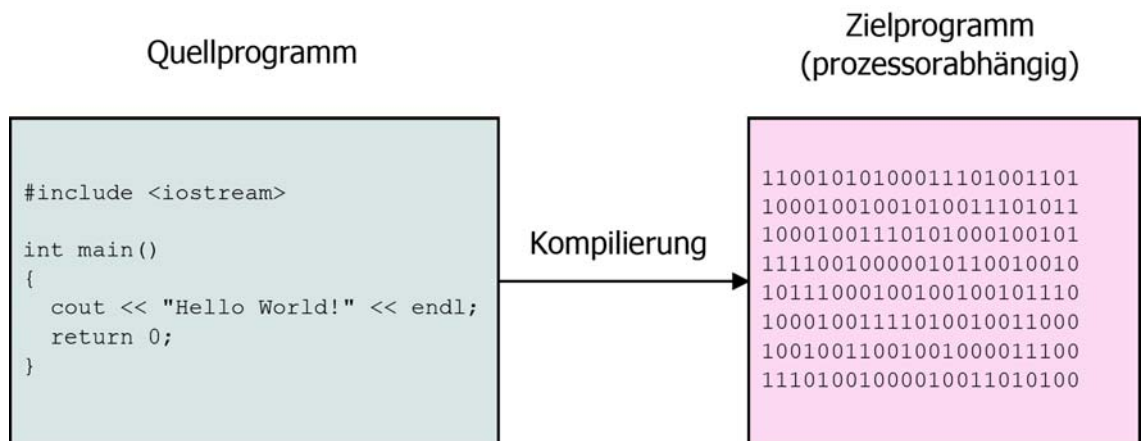


Abbildung 2: Kompilierungsvorgang, vom Quellcode zum ausführbaren Programm

Um den C++-Quellcode in den Binärcode umzuwandeln, benötigen wir einen sog. Compiler. Unter dem Betriebssystem Linux hat sich die GCC (GNU Compiler Collection) seit Jahrzehnten etabliert. Unser `Hello World`-Quellcode kann mit Hilfe der GCC in ein ausführbares Binärprogramm umgewandelt werden.

```
g++ -o hello hello.cpp
```

- `g++`: Das C++-Compilerprogramm
- `-o hello`: Name des ausführbaren Programms (output)

- `hello.cpp`: Name der Quellcode-Datei

Nach dem Kompilierungsvorgang existiert im selben Verzeichnis das ausführbare Programm `hello`. Nach dessen Aufruf muss `Hello World` auf dem Bildschirm erscheinen (`./hello`).

3.3 Aufgabe des C++-Compilers

Beim Übersetzungsvorgang hat der Compiler die im Folgenden genannten Aufgaben:

1. **Lexikalische Analyse:** Zerteilen des Quellcodes in sog. Tokens verschiedener Typen (Schlüsselwörter, Bezeichner, Zahlen und Zeichenketten)
2. **Syntaktische Analyse:** Überprüfung der Struktur des Quellcodes. Struktur muss kontextfreie Syntax der Quellsprache (C++) entsprechen.
3. **Semantische Analyse:** Überprüfung der statischen Bedingungen an das Programm. Variablen müssen deklariert werden, bevor sie genutzt werden können.

Findet der Compiler einen Fehler, wird die Kompilierung abgebrochen. Der Compiler gibt die Fehlermeldungen mit Zeilennummern aus. So hat es der Softwareentwickler leichter, den Fehler im Quellcode zu finden. Der Kompilierungsprozess für unser `hello.cpp` kann etwas detaillierter dargestellt werden:

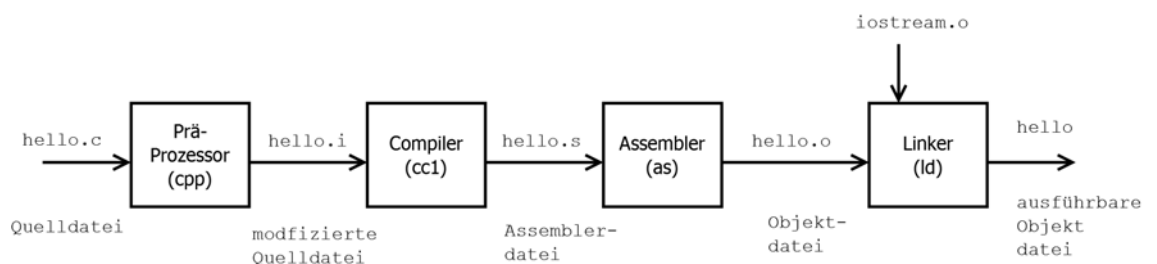
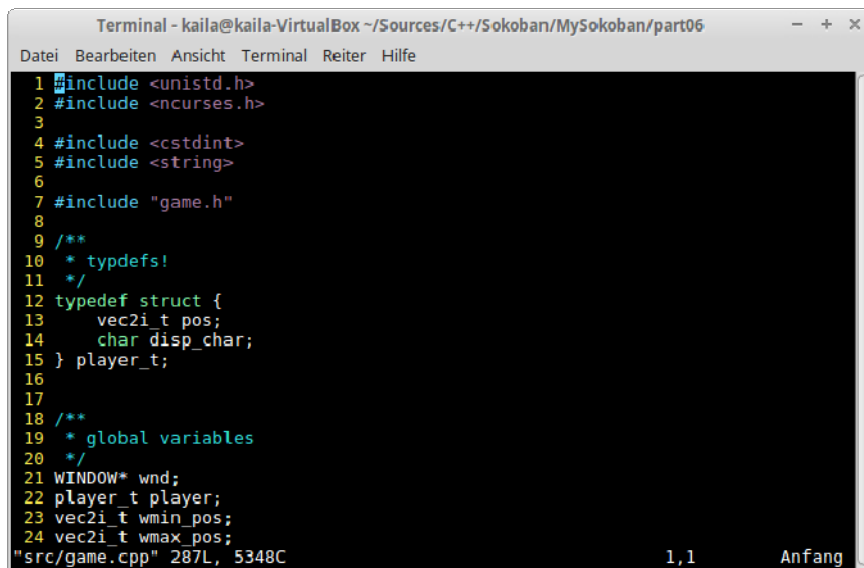


Abbildung 3: detaillierter Kompilierungsvorgang unseres "Hello World"-Programms

3.4 Der Editor vi für die Eingabe des C++-Quellcodes

Linux beinhaltet als Standardeditor den `vi`. Er ist auf allen Unix- und Linux-Rechnern verfügbar. Für die Benutzung des Editors ist kein X-Window nötig, denn `vi` läuft auf nahezu jedem Terminal. Modifizierte Variante: `vim`.



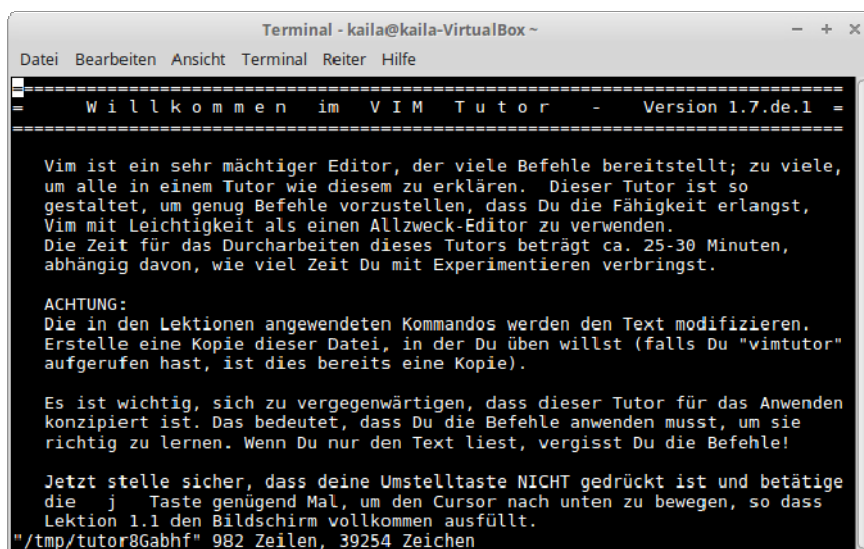
```

Terminal - kaila@kaila-VirtualBox ~/Sources/C++/Sokoban/MySokoban/part06
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
1 #include <unistd.h>
2 #include <ncurses.h>
3
4 #include <cstdlib>
5 #include <string>
6
7 #include "game.h"
8
9 /**
10  * typedefs!
11  */
12 typedef struct {
13     vec2i_t pos;
14     char disp_char;
15 } player_t;
16
17
18 /**
19  * global variables
20  */
21 WINDOW* wnd;
22 player_t player;
23 vec2i_t wmin_pos;
24 vec2i_t wmax_pos;
"src/game.cpp" 287L, 5348C          1,1          Anfang

```

Abbildung 4: Der vi in der Konsole

Da der `vi` eine sehr kryptisch anmutende Tastatursteuerung hat, ist es mit ihm zu Anfang recht schwierig, Code zu schreiben. Hat man sich an das Interface gewöhnt, kann man sehr schnell und effizient arbeiten. Zum Einstieg in den `vi` oder `vim` empfiehlt sich das vim-Tutorial `vimtutor`. Es übt die wichtigsten Tastaturbefehle ein und bietet somit ein sog. Walkthrough durch den Editor.



```

Terminal - kaila@kaila-VirtualBox ~
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
=====
=  W i l l k o m m e n  i m  V I M  T u t o r  -  V e r s i o n  1.7.d e .1  =
=====

Vim ist ein sehr mächtiger Editor, der viele Befehle bereitstellt; zu viele,
um alle in einem Tutor wie diesem zu erklären. Dieser Tutor ist so
gestaltet, um genug Befehle vorzustellen, dass Du die Fähigkeit erlangst,
Vim mit Leichtigkeit als einen Allzweck-Editor zu verwenden.
Die Zeit für das Durcharbeiten dieses Tutors beträgt ca. 25-30 Minuten,
abhängig davon, wie viel Zeit Du mit Experimentieren verbringst.

ACHTUNG:
Die in den Lektionen angewendeten Kommandos werden den Text modifizieren.
Erstelle eine Kopie dieser Datei, in der Du üben willst (falls Du "vimtutor"
aufgerufen hast, ist dies bereits eine Kopie).

Es ist wichtig, sich zu vergegenwärtigen, dass dieser Tutor für das Anwenden
konzipiert ist. Das bedeutet, dass Du die Befehle anwenden musst, um sie
richtig zu lernen. Wenn Du nur den Text liest, vergisst Du die Befehle!

Jetzt stelle sicher, dass deine Umstelltaste NICHT gedrückt ist und betätige
die j Taste genügend Mal, um den Cursor nach unten zu bewegen, so dass
Lektion 1.1 den Bildschirm vollkommen ausfüllt.
"/tmp/tutor8Gabhf" 982 Zeilen, 39254 Zeichen

```

Abbildung 5: Walkthrough vimtutor

3.5 C/C++-Style-Guidelines

Ein Style-Guide legt Regeln fest, die vor allem dazu dienen sollen, die Lesbarkeit und Wartbarkeit des Codes zu erhöhen. In Unternehmen entstehen

häufig für die Wartung von Software größere Kosten als für deren Erstellung, und der weitaus größte Teil des Codes wird öfter gelesen als geschrieben. In Unternehmen werden daher i.d.R. umfangreiche Style-Guidelines für die verschiedenen Programmiersprachen angewandt. Mit diesem kleinen Style-Guide können Sie sich einige Aspekte eines verbreiteten Stils angewöhnen und die Einhaltung von Vorgaben üben, wie sie in der beruflichen Praxis vorkommen. Beachten Sie die zu diesem Style-Guide gehörenden Dateien **cstyle.c** und **cstyle.h**, die exemplarische Beispiele zu den Regeln enthalten.

Diese Regeln sind im Unterricht zwingend anzuwenden!

4. Sokoban - Das Computerspiel

Sokoban ist ein altes Computerspiel von Hiroyuki Imabayashi, der es um 1982 entwickelt und über seine Firma **Thinking-Rabbit** erstmals für verschiedene Computersysteme vertrieben hat. Sokoban (倉庫番) bedeutet auf Japanisch **Lagerhausverwalter** und genau darum geht es in dem Spiel. Die ersten in Europa erhältlichen Sokoban-Versionen kamen aus dem Hause Spectrum HoloByte (1984, z.B. für Apple II E).

4.1 Spieleprinzip

In den meisten Umsetzungen muss der Spieler eine kleine Figur, den Sokoban, steuern (*Moves*) und mit ihm die verstreuten Kisten mit möglichst wenigen Verschiebungen (*Pushes*) an ihre vorgesehenen Plätze im Lagerhaus schieben. Das Lagerhaus ist häufig sehr verwinkelt und kompliziert aufgebaut.

Der Sokoban kann immer nur eine Kiste vorwärts bewegen; er kann eine Kiste auf keinen Fall aus einer Ecke und nur unter bestimmten Bedingungen von einer Wand entfernen. Der Sokoban kann die Kisten nicht ziehen oder anders bewegen, er muss die Kisten schieben.

Das sieht auf den ersten Blick oft sehr einfach aus, entpuppt sich aber meistens als äußerst knifflige Angelegenheit, da man leicht den Weg des Sokoban verstellen oder eine Kiste in eine Stellung bringen kann, in der sie nicht mehr zu bewegen ist. So muss man oft Kisten in die Gegenrichtung schieben und neu sortieren, bevor man eine Kiste nach der anderen an die markierten Positionen schieben kann. Der Schwierigkeitsgrad bei sehr schweren Lagerhäusern kommt einem Schachspiel sehr nah. Der Anspruch dieses simplen Spielprinzips ist also sehr hoch.

4.2 Lagerhaus-Design

Eine ebenso große Herausforderung wie die Aufgaben zu lösen ist es, solche Aufgaben zu erfinden. Neben dem Erfinder des Spiels haben mittlerweile eine große Zahl von Autoren **Aufgaben**, auch Level genannt, für den Lagerarbeiter geschaffen. Das Original-Spiel hatte 50 verschiedene Aufgaben mit

unterschiedlichen Schwierigkeitsgraden; etwas später gab es ein weiteres Paket mit 40 neuen Aufgaben. Einen guten Überblick über bisher veröffentlichte Aufgaben-Sets gibt <http://www.sourcecode.se/sokoban/levels.php> oder <http://sokobano.de/de/levels.php>. Dort kann man sich die Aufgaben auch in verschiedenen Formaten ansehen, herunterladen und von dort die Webseiten zahlreicher Autoren besuchen.

Die Aufgaben für den Sokoban folgen stets einem speziellen Format. Das meist genutzte Format ist folgendermaßen festgelegt:

	Zeichen	Bedeutung
<code>; Map1</code>	<code>;</code>	gibt ein Kommentar, meist Name der Aufgabe
<code>#####</code>	<code>#</code>	ein Wand-Teil für das Lagerhaus
<code># \$ #</code>	<code>\$</code>	eine Kiste
<code># \$ \$ #</code>	<code>.</code>	eine Zielposition für eine Kiste
<code># ###</code>	<code>@</code>	den Spieler
<code>####\$ @#</code>		
<code>#.. . ###</code>		
<code>#####</code>		

Abbildung 6: Aufgaben-Format

Neue Aufgaben können über einen Programmiereditor z.B. den `vi` oder `vim` sehr einfach erstellt werden.

5. Design und Entwicklung des Computerspiels Sokoban

Dieser Abschnitt ist für die ALLE Studenten gedacht und betrifft die Basis des Computerspiels Sokoban!

Die Spielregeln für Sokoban sind nicht besonders schwer. Deshalb ist es recht einfach, die Regeln in ein Computerprogramm umzusetzen. Es sind zwei grundsätzliche Aspekte zu betrachten, zum einen die grafische Repräsentation des Spiels und zum anderen die Umsetzung des Spieleautomaten mit der ganzen Logik für die Spielregeln. Zuerst widmen wir uns der grafischen Ein- und Ausgabe von Sokoban. Der Einfachheit halber soll das Spiel in einer Konsole lauffähig sein. Das hat den Vorteil, dass es über SSH auf einem sog. Headless-Linux-System, z.B. Raspbian, gespielt werden kann. Das User-Interface solcher Anwendungen nennt man TUI, Text User Interface. Nachfolgend dargestellt ist das fertige Spiel in einer Konsole:



Abbildung 7: Das Konsole-Spiel Sokoban

Zum Vergleich eine Sokoban-Umsetzung mit Java und Swing:

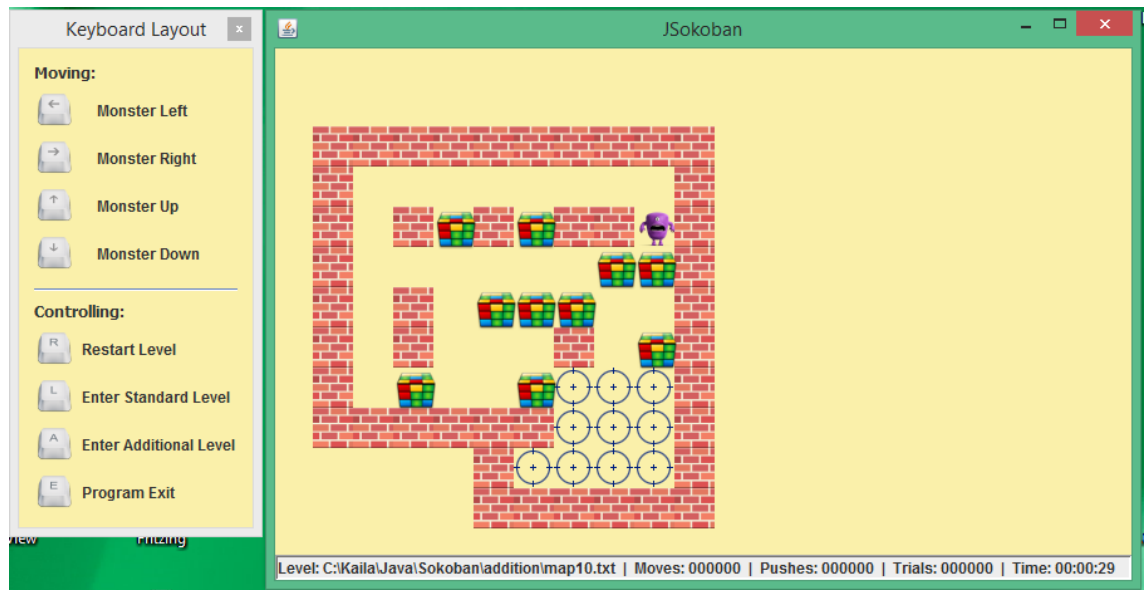


Abbildung 8: Eine Java-Swing-Umsetzung des Spiels Sokoban

5.1 Grafische Ausgabe mit der Ncurses-Bibliothek

Ncurses ist eine C/C++-Bibliothek für die Steuerung von Textterminals. Hauptzweck dieser Bibliothek ist die Erstellung von TUIs (Text User Interface). Typische Beispiele für Programme, deren Benutzeroberflächen die Ncurses-Bibliothek benutzen, sind der Lynx-Browser, der GNU Midnight Commander, das Linux-Kernel-Konfigurationsprogramm in der Ncurses-Variante oder das Ncurses-Frontend des YaST-Installations- und -Konfigurationsprogramms bei der SuSE-Linux-Distribution. Ncurses ist für diverse Unix-Plattformen erhältlich und steht unter der MIT-Lizenz.

Die Installation der Ncurses-Bibliothek ist in der einschlägigen Literatur beschrieben und wird an dieser Stelle vorausgesetzt. Sämtliche Funktionen der Bibliothek werden in den nachfolgenden Parts angesprochen.

Eine Studie der API-Beschreibung ist ein Pflichtprogramm!

5.2 Part 00 – Start mit Standard-Ein- und Ausgaben

Angenommen es existiert folgender Pfad `~/part00`, dann legen Sie dort die Datei `main.cpp` an. Das Beispiel liest einen kompletten String inkl. Leerzeichen und ein Integer-Wert ein und gibt beide Eingaben auf den Monitor aus.

```

1 /**
2  * Part 0-1
3  */
4
5 #include <iostream>
6
7 using namespace std;
8
9 int main()
10 {
11     string str;
12     int i;
13
14     cout << "Bitte einen String eingeben: ";
15     getline(cin, str);
16     cout << "Bitte einen Integer eingeben: ";
17     cin >> i;
18     cout << "String: " << str << " Integer: " << i << endl;
19
20     return 0;
21 }

```

Zeile	Funktion
7	Definition des Namenraums (<code>namespace</code>). Damit kann bei allen <code>std</code> -Methoden das <code>std::</code> wegfallen. Das war in Abbildung 1 noch nötig.
11	Deklaration einer <code>string</code> -Variable <code>str</code> , später mehr
12	Deklaration einer <code>integer</code> -Variable <code>i</code> , später mehr
15	Die <code>getline()</code> -Methode liest eine komplette Eingabe (<code>cin</code>) inkl. Leerzeichen in einen String <code>str</code> ein.
17	Die <code>cin</code> -Methode liest eine Eingabe bis zum ersten Leerzeichen ein. Die Eingaben können auch ganze oder Fließkommazahlen sein.

Zum Übersetzen in ein ausführbares Programm in `~/part00` geben Sie folgendes in der Konsole ein:

```
~/part00$ g++ -o main main.cpp -g -std=c++17 -Wall
```

Zeile	Funktion
1	<ul style="list-style-type: none"> • <code>g++</code>: Der C++-Compiler • <code>-o name</code>: Name des Programms • <code>-g</code>: Debug-Informationen einfügen • <code>-std=c++17</code>: Übersetzen nach C++17-Standard • <code>-Wall</code>: Alle Compiler-Warnungen einschalten

g++ im Verzeichnis ~/part01 erzeugt ausführbare Datei

Das Programm `main` ist nun in `~/part00` vorhanden und ausführbar.

Analysieren Sie die Eingaben und die dazugehörigen Ausgaben.

5.3 Part 01 – Start mit der Ncurses-Bibliothek

Erzeugen Sie folgenden Pfad `~/part01`. Legen Sie dort die Datei `main.cpp` an. Der Inhalt der Datei ist:

```
1 /**
2  * Part 1-2
3  */
4
5 #include <ncurses.h>
6
7 int main(int argv, char **argv)
8 {
9     initscr();    // NCurses: Bildschirm initialisieren
10    cbreak();     // NCurses: Strg+C zum Abbrechen einschalten
11    noecho();     // NCurses: Zeichen-Echo ausschalten
12    clear();      // NCurses: Bildschirm löschen
13    refresh();    // NCurses: Bildschirm neu zeichnen
14
15    while(true);
16
17    return 0;
18 }
```

Die Ncurses-Funktionen in Zeile 9-13 bewirken das, was als Kommentar dahinter steht.

Neu ist das Schlüsselwort `while` in Zeile 15.

5.3.A Endlosschleife mit `while`, kopfgesteuert

Eine Schleife wird verwendet, um Wiederholungen im Programm zu realisieren. Jede Schleife hat eine Schleifenbedingung, die vor einem Durchlauf geprüft wird. Ist die Bedingung erfüllt, werden die Befehle innerhalb der Schleife ausgeführt. Ist die Bedingung nicht erfüllt, wird die Schleife verlassen. Eine Endlosschleife ist eine besondere Schleife, in der die Bedingung **immer** erfüllt ist. Im obigen Beispiel in Zeile 15 sehen Sie eine Endlosschleife. Die Bedingung steht hinter den Schlüsselwort **while** in runden Klammern und lautet hier **true**, immer erfüllt. Häufig befindet sich dort auch eine 1 oder eine andere Zahl. Das

ist gleichzusetzen mit `true`. Die einzige Zahl, die mit `false` gleichzusetzen ist, ist die Zahl 0.

Bedingung erfüllt	Bedingung nicht erfüllt
<code>true</code> , 1, 5, 102234, -43, ...	<code>false</code> , 0

5.3.B Hallo Ncurses, string, auto und range-based for

Jetzt implementieren Sie folgenden Quellcode vor die Endlosschleife.

```

1 move(5, 5);
2
3 std::string text = "Hello Ncurses!"
4 for(auto ch :text)
5 {
6     addch(ch);
7     addch(' ');
8 }
9
10 refresh();

```

Zeile	Funktion
1	<code>move</code> bewegt den Cursor auf die angegebene Position <code>y/x</code> .
3	Deklaration einer Stringvariablen <code>text</code> mit gleichzeitiger Initialisierung mit <code>"Hello Ncurses!"</code> .
4	range-based for-Schleife mit einer Deklaration einer auto-Variablen. Schleife läuft vom ersten Zeichen im String bis zum letzten Zeichen. Jedes Zeichen pro Schleifendurchlauf ist in <code>ch</code> .
5-8	Schleifenblock, der durchlaufen wird, solange die Bedingung <code>true</code> ist.
6,7	Ausgabe eines Zeichens aus der Stringvariablen <code>text</code> mit einem nachfolgenden Leerzeichen.
10	Ausgabe auf den Bildschirm mit <code>refresh</code> veranlassen.

Der Text `"H e l l o N c u r s e s !"` wird nun auf dem Bildschirm bei den angegebenen Koordinaten geschrieben.

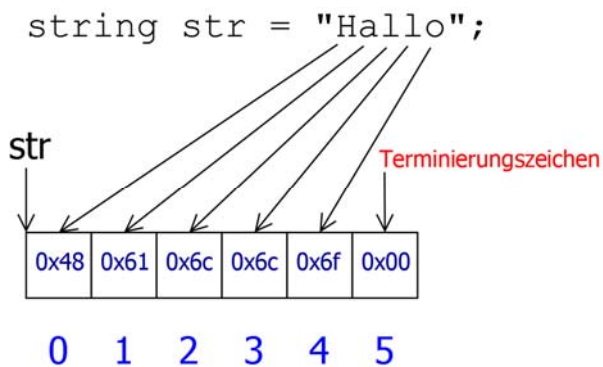
5.3.B.1 Der Datentyp string

Der Datentyp `string` in Zeile 3 stellt ein Container für Zeichen dar. Es handelt sich hier um eine Zeichenkette. Es folgt eine kurze Darstellung des Datentyps, eine vollständige Beschreibung finden Sie in im C++11-Standard.

Methode/-Operator	Funktion
-------------------	----------

<code>at(pos)</code>	Gibt Zeichen an Position <code>pos</code> eines Strings zurück. <code>pos=0</code> heißt: erstes Zeichen!
<code>length()</code>	Gibt die Länge eines Strings zurück
<code>clear()</code>	Leert einen String
<code>data()</code> <code>c_str()</code>	Konvertiert ein C++-String in eine C-Zeichenkette. Ist wichtig, wenn C-Funktionen angewendet werden
<code>+</code>	Fügt Strings zusammen

Das erste Zeichen Strings beginnt immer bei 0 und endet beim sog. Nullterminierungszeichen `'\0'` oder `0`. Das spezielle Zeichen gibt immer das Ende eines Strings an. Man kann sich ein String `Hallo` folgendermaßen vorstellen:



Der String `str` hat 6 Bytes im Speicher reserviert, 5 Bytes für den String als solcher und das letzte Byte für die Terminierung.

Der Datentyp `string` ist vollkommen dynamisch. Eine Größenangabe ist nicht mehr nötig.

Wenn Strings vergrößert oder verkleinert werden, wird die Größe des Strings dynamisch angepasst.

5.3.B.2 Der Datentyp auto

Der `auto`-Datentyp ist ein spezieller Datentyp. Der Compiler kann während des Übersetzungsvorgangs anhand der Daten feststellen, welcher Datentyp oder Container genommen werden muss. Ergo muss bei der Deklaration einer Variablen des Datentyps `auto` eine Initialisierung folgen.

<code>auto var;</code>	Nicht korrekt, Compiler kann Datentyp nicht bestimmen
<code>auto var="Hello"</code>	Datentyp <code>string</code>
<code>auto var=76;</code>	Datentyp <code>int</code>
<code>auto var=2.67;</code>	Datentyp <code>float</code>

Der Datentyp wird auch gerne verwendet, um auf Container-Objekte zuzugreifen. Dazu an geeigneter Stelle mehr.

5.3.B.3 Die Range-based for-Schleife

Die Range-based for-Schleife ist besonders im Umgang mit Containern vom

Typ `string`, `vector`, `list` und `map` interessant und steht ab dem C++11-Standard zur Verfügung. In Zeile 4 erkennt man den Aufbau. Vor dem `:` wird ein einzelnes Element aus dem Container je nach Schleifendurchlauf zur Verfügung gestellt. Nach dem `:` steht der ganze Container.

5.3.C Das Ncurses-Koordinatensystem

Wenn die Ncurses-Bibliothek genutzt wird, ist der sichtbare Bildschirm genau festgelegt. Der Bildschirm gleicht einem Koordinatensystem in der Geometrie. Es ist ein 2-dimensionales Koordinatensystem, die Spalten (Columns) sind auf der X-Achse und die Zeilen (Rows) auf der Y-Achse festgelegt wie folgt:

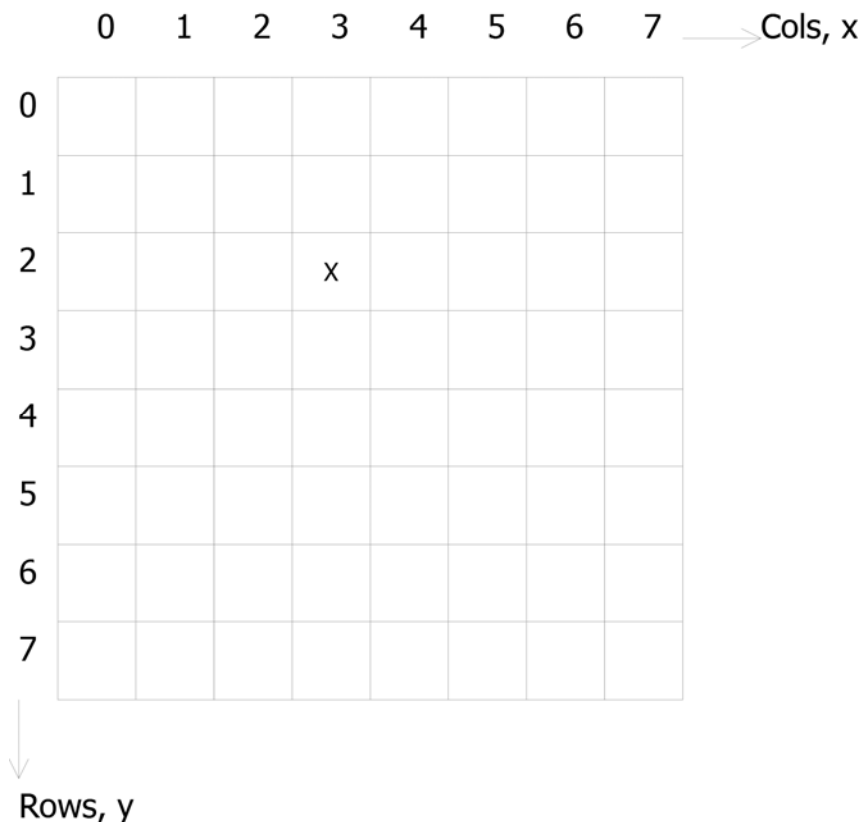


Abbildung 9: Koordinatensystem in Ncurses

Das Zeichen `x` ist an der Position `y=2, x=3` zu finden. Alle Funktionen in Ncurses, die eine Position benötigen, beruhen auf dieses Koordinatensystem. I.d.R. haben Terminalfenster eine Auflösung von 80x24 Blockpunkte.

`move(5,5)` bewegt den Cursor auf die entsprechende Position

Ein Pixel:

Der Begriff Pixel ist ein Kunstwort aus dem englischen Picture und Element und bezeichnet den kleinsten Bildpunkt eines 2-dimensionalen Koordinatensystems. In Abbildung 9 wäre es ein Viereck mit einer x - und y -Koordinate, in dem ein Zeichen dargestellt werden kann.

5.3.D Triggerpunkt Part 01

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.4 Part 02 – Ein farbiges Ncurses-Fenster mit Eingaben

Nun gilt es, einen Spieler in einem definierten Bereich zu platzieren und ihn über die Tastatur zu steuern. Eigentlich eine simple Aufgabe, die aber nicht ganz ohne ist. Wie wird gesteuert? Der geneigte Spieler weiß Aufgrund seiner langjährigen Erfahrung mit `Doom`, dass die Tasten `w`, `a`, `d` und `s` sich für die Steuerung erprobt haben. Die Pfeiltasten sollen aber auch implementiert werden, so dass folgende Tastaturbelegung sinnvoll ist:









Tasten	Bedeutung
 	Spieler bewegt sich nach Norden
 	Spieler bewegt sich nach Süden
 	Spieler bewegt sich nach Westen
 	Spieler bewegt sich nach Osten

Abbildung 10: Tastaturbelegung für die Bewegung des Spielers

Aber, zuerst einmal ein neuer Aufbau des Programms.

5.4.A Altes Programm, neuer Aufbau

Wir werden nun des besseren Aufbaus wegen ein neues Programmdesign in `~/part02` erstellen. Dazu werden 3 neue Funktionen in `main.cpp` implementiert:

```
1 int init();
2 void run();
3 void close();
```

Zeile	Funktion
1,2,3	Alle selbst entwickelten Funktionen können über die <code>main</code> -Funktion implementiert werden. Werden Sie darunter implementiert, müssen sog.

Prototypen erstellt werden.

Die init-Funktion:

```

1 int init() {
2   wnd = initscr();      // Ncurses: Initialisierung mit wnd-Handler
3   cbreak();           // siehe Part01
4   noecho();
5   clear();
6
7   keypad(wnd, true);   // Sondertasten ein, für Spielersteuerung
8
9   curs_set(0);        // Cursor aus, würde sonst am Spieler blinken
10
11  start_color();       // Farben einschalten
12
13  init_pair(1, COLOR_WHITE, COLOR_BLUE);
14  bkgd(COLOR_PAIR(1)); // Hintergrund/Vordergrundfarben des Fensters
15
16  attron(A_BOLD);      // Attribut Fett ein
17  box(wnd, 0, 0);      // Box über gesamten Bildschirm zeichnen
18  attroff(A_BOLD);     // Attribut Fett aus
19
20  return 0;
21 }
```

Zeile	Funktion
1	Eine Funktionsdefinition mit Kopf <code>int init()</code> und Rumpf <code>{...}</code>
2-18	Siehe Kommentare
20	Rückgabewert der Funktion ist 0

Die run-Funktion

```

1 void run() {
2   bool exit_requested = false;
3
4   while(!exit_requested) {
5     int in_char = wgetch(wnd);
6     switch(in_char) {
7       case 'q': // quit game
8         exit_requested = true;
9         break;
10      case KEY_UP: // up
11        case 'w':
12          break;
13      case KEY_DOWN: // down
14        case 's':
15          break;
16      case KEY_LEFT: // left
17        case 'a':
18          break;
19      case KEY_RIGHT: // right
20        case 'd':
21          break;
22      default:
23        break;
24    }
25    usleep(WAIT_TICK); // 10 ms

```

```

26     refresh();
27     }
28 }

```

Zeile	Funktion
1	Funktionskopf <code>void run()</code> und Rumpf {...}
2	Boolsche Variablendeklaration für das Abbrechen der <code>while</code> -Schleife
4	<code>while</code> -Schleife: Solange Variable nicht wahr, führe Rumpf aus {...}
5	Funktion <code>wgetch</code> wartet solange, bis der Nutzer ein Zeichen eingibt. Das wird in der Variablen <code>in_char</code> gespeichert
6	<code>switch</code> -Anweisung: Wertet die eingegebenen Zeichen aus (<code>in_char</code>)
7,8,9	Case-Block für <code>'q'</code> : setze boolsche Variable so, dass die <code>while</code> -Schleife abgebrochen wird
10-12	Case-Block für <code>'w'</code> oder <code>KEY_UP</code> : Steuerung des Spielers nach Norden
13-21	Siehe 10,11,12: nur in die anderen Richtungen
25	Programm wartet ca. 10ms an dieser Stelle, macht danach weiter
26	<code>Refresh</code> sorgt für das neue Zeichnen des Fensters, um die Änderungen anzuzeigen

Die `close`-Funktion

```

1 void close() {
2     endwin();
3 }

```

Zeile	Funktion
1	Eine Funktionsdefinition mit Kopf <code>int close()</code> und Rumpf {...}
2	<code>Endwin</code> beendet die mit <code>initscr</code> installierte <code>Ncurses</code> -Initialisierung

Die `main`-Funktion:

```

1 #include <string>
2 #include <unistd.h>
3 #include <ncurses.h>
4
5 #define WAIT_TICK 10000
6
7 WINDOW *wnd;
8
9 int init() {...}
10 void run() {...}
11 void close() {...}
12

```

```

13 int main () {
14     init();
15     run();
16     close();
17 }

```

Zeile	Funktion
1-3	Die Verweise auf <code>string-</code> <code>unistd-</code> und <code>ncurses-</code> Bibliotheken
5	Definition einer Konstanten <code>WAIT_TICK</code> mit dem Wert <code>10000</code>
7	Globale Variablendeklaration <code>wnd</code> vom Datentyp <code>WINDOW *</code> . Wird für <code>Ncurses-</code> Initialisierung benötigt.
9-11	Die Funktionsdefinitionen wie oben beschrieben
13	Main-Funktionskopf
14-16	Main-Funktionsrumpf: Aufruf der Funktionen <code>init()</code> , <code>run()</code> und <code>close()</code> . Hier werden die Funktionen tatsächlich ausgeführt!

Um den Kompilierungsvorgang zu vereinfachen, wird ein einfaches `Makefile` `~/part02` erstellt, das die Kompilierung nach Anweisung durchführt. An dieser Stelle wird der Aufbau des `Makefiles` auf später verschoben:

```

1 CXX = g++
2 CXXFLAGS = -g -std=c++17 -Wall -pedantic
3 LDXXFLAGS = -lncurses
4
5 OBJS = main.o # hier werden zukünftige Module hinzugefügt!
6
7 prog: $(OBJS) # Achtung: Einrücken mit Tabs, nicht mit Leerzeichen!
8             $(CXX) $(CXXFLAGS) -o prog $(OBJS) $(LDXXFLAGS)
9
10 %.o: %.cpp
11         $(CXX) $(CXXFLAGS) -c $<
12
13 clean:
14         rm -f prog *.o

```

Der Aufruf `make` erzeugt das ausführbare Programm `prog`

5.4.A.1 Der Datentyp int

Der Datentyp `int` (Integer) ist in der Lage, ganzzahlige Werte zu speichern. Nach Standard ist er mindestens 16 Bit, also 2 Byte breit. Auf 32 oder 64 Bit Betriebssystemen ist dieser Datentyp mindestens 32 Bit, also 4 Byte breit. Wird der Datentyp als solcher verwendet, ist er `signed`, also vorzeichenbehaftet. Stellt man das Schlüsselwort `unsigned` dafür, ist er vorzeichenlos.

Bits	Format	Wertebereich
16	(signed) int	$\pm 3.27 \cdot 10^4$
	unsigned int	0 to $6.55 \cdot 10^4$
32	(signed) int	$\pm 2.14 \cdot 10^9$
	unsigned int	0 to $4.29 \cdot 10^9$
64	(signed) int	$\pm 9.22 \cdot 10^{18}$
	unsigned int	0 to $1.84 \cdot 10^{19}$

5.4.A.2 Der Datentyp bool

Der Datentyp `bool` kennt nur zwei Zustände: `true` oder `false`. Für `true` kann auch 1 und für `false` auch 0 geschrieben werden. Boolesche Variablen werden häufig verwendet, um zu prüfen, ob einmalige Ereignisse aufgetreten sind oder nicht. So ein Ereignis ist z.B. die Taste ‚q‘, um das Programm zu beenden.

Bits	Format	Wertebereich
8	bool	True, 1
		False, 0

5.4.A.3 Funktionsdefinition und Funktionsaufruf

Um Ordnung in ein Programm zu bringen, werden große Probleme in kleine Teilprobleme zerlegt. Diese Teilprobleme werden dann als Funktion getrennt in Modulen definiert und können dann in anderen Funktionen aufgerufen und ausgeführt werden. Eine Methode erledigt immer eine bestimmte Aufgabe.

```

1 int init() {
2     wnd = initscr();
3     ...
21 }
```

Die Funktion `int init()` besteht aus dem Funktionskopf und dem Funktionsrumpf.

Funktionskopf `int init()`:

Der Funktionskopf besteht aus dem Funktionsnamen `init`. Vor dem Namen steht der Datentyp des Rückgabewerts `int`. Es können alle einfachen und komplexen Datentypen verwendet werden. In den Klammern hinter dem Namen stehen ggf. Parameter, die der Funktion übergeben werden. Sind keine

Parameter vorhanden, müssen trotzdem die Klammern leer gesetzt werden ().

Funktionsrumpf {...}:

Im Funktionsrumpf { } stehen die Aktionen, die die Funktion ausführen soll.

Zum Schluss des Blocks wird der Rückgabewert mit `return` zurückgegeben.

```

1 int init() {
2     ...
20    return 0;    // Rückgabewert 0
21 }
```

Wenn eine Funktion kein Rückgabewert besitzt, wird das mit dem Schlüsselwort `void` im Funktionskopf angegeben. Im Funktionsrumpf kann dann entweder `return;` (ohne Wert) oder das `return` weggelassen werden.

5.4.A.4 Die while-Schleife

Die while-Schleife ist der in 5.3.A beschriebenen Endlosschleife ähnlich. Nur die Bedingung innerhalb der beiden Klammern (. . .) ist `variable`. Die Bedingung kann mit Vergleichsoperatoren und / oder mit Logikoperatoren kombiniert werden. Vergleichsoperatoren werden genutzt, um Variablen mit festen Werten oder anderen Variablen zu vergleichen. Logische Operatoren nutzen boolesche Verknüpfungen für Bedingungen.

Bedingungen werden bei while-/do-while-/for-Schleifen und if-Verzweigungen verwendet.

5.4.A.4.1 Bedingungen in Verzweigungen und Schleifen

Vergleichsoperatoren:

Operator	Wirkung
<code>==</code>	gleich
<code>!=</code>	ungleich
<code>></code>	größer als
<code>>=</code>	größer oder gleich
<code><</code>	kleiner
<code><=</code>	Kleiner oder gleich

Logische Operatoren:

Operator	Wirkung
&&	UND-Verknüpfung
	ODER-Verknüpfung
!	Negierung

Nachfolgend werden einige Bedingungen dargestellt und erörtert. Es kommt häufig vor, dass Vergleichsoperatoren mit logischen Verknüpfungen kombiniert werden. Dies führt dann u.U. zu komplexen Bedingungen, die dann schriftlich aufgelöst werden müssen.

Beispiele	Wirkung
<code>while(4711) { }</code>	Wahr, while-Schleife wird ausgeführt
<code>while (-67) { }</code>	Wahr, while-Schleife wird ausgeführt
<code>while (0) { }</code>	Unwahr, while-Schleife wird nicht ausgeführt
<code>while (a==0) { }</code>	Wahr, wenn a gleich 0, while-Schleife wird ausgeführt
<code>while (a!=0) { }</code>	Wahr, wenn a ungleich 0, while-Schleife wird ausgeführt
<code>while (a==0 && b!=0) { }</code>	Wahr, wenn a gleich 0 und b ungleich 0, while-Schleife wird ausgeführt
<code>while (!a) { }</code>	Wahr, wenn a nicht 0, while-Schleife wird ausgeführt
<code>while (!a b==true) { }</code>	Wahr, wenn a nicht 0 oder b gleich true, while-Schleife wird ausgeführt

Bei den doppelten Gleichheitszeichen kann es passieren, dass eines vergessen wird. Dann entstehen 2 Fehler:

```
while (exit_requested = false)
```

- Aus der Bedingung wird eine Zuweisung, `exit_requested` führt nach der Abarbeitung der Zeile den Wert `false`
- Die Zuweisung ist für `while` trotzdem eine Bedingung, das Ergebnis wird ausgewertet, hier also unwahr (`false`)

5.4.A.5 Verzweigungen mit switch-case

Ist es erforderlich, viele Verzweigungsfälle zu unterscheiden und für jeden Fall unterschiedliche Aktionen auszuführen, so kann das mit einer `switch-case`-Anweisung erreicht werden. In den Klammern nach dem Schlüsselwort `switch` muss der Ausdruck stehen, der ausgewertet wird. In vielen Fällen ist dieser Ausdruck eine Variable. Danach folgen mit dem Schlüsselwort `case` die verschiedenen Fälle, nach dem Doppelpunkt die auszuführenden Befehle. Der `case`-Block wird mit `break` abgeschlossen. Dies ist unbedingt notwendig. Mit

`break` wird bei erfolgreichem Ausführen eines Falles die `switch`-Anweisung verlassen. Wird keines der Fälle erreicht, wird der `default`-Block ausgeführt. Ein besonderer Anwendungsfall ist es, wenn mehrere Fallunterscheidungen zusammengeführt werden können. Dann setzt man bei der letzten Fallunterscheidung das `break`.

```
19     case KEY_RIGHT:    // Bei Taste KEY_RIGHT
20     case 'd':         // oder bei Taste 'd'
21         ...           // mache etwas
22         break;        // bis hierher
```

5.4.B Farben und Boxen mit Ncurses

Farben müssen in Ncurses zuerst einmal eingeschaltet werden. Danach müssen Farbpaare gebildet werden, die mit einer eindeutigen Kennung belegt werden. Dann können die Farbpaare bestimmten Zeichenfunktionen zugewiesen werden.

```
1 start_color();
2 init_pair(1, COLOR_WHITE, COLOR_BLUE);
3 wbkgd(wnd, COLOR_PAIR(1));
```

Zeile	Funktion
1	Start Farbwahl mit <code>start_color</code> . Muss einmal aufgerufen werden!
2	Definition des Farbpaares <code>init_pair</code> mit eindeutiger Kennung: <ul style="list-style-type: none"> - Kennung: 1 - Schriftfarbe: Weiß - Hintergrundfarbe: Blau
3	Nutzung des Farbpaares auf ein Fenster (<code>wnd</code>) mit <code>wbkgd</code>

Boxen werden folgendermaßen gezeichnet (später nicht mehr nötig):

```
1 attron(A_BOLD);
2 box(wnd, 0, 0);
3 attroff(A_BOLD);
```

Zeile	Funktion
1	Mit <code>attron</code> (attribute on) werden Attribute wie <code>bold</code> eingeschaltet
2	Mit <code>box</code> wird innerhalb des Fensters <code>wnd</code> ein Rahmen gezeichnet. Sind die beiden anderen Parameter 0, wird der Rahmen über das gesamte

	Fenster gespannt.
3	Mit <code>attoff</code> werden Attribute wieder ausgeschaltet

5.4.C Das Resultat

Nachdem Part 02 soweit verstanden und implementiert ist, sollte das Programm folgendes Aussehen haben:

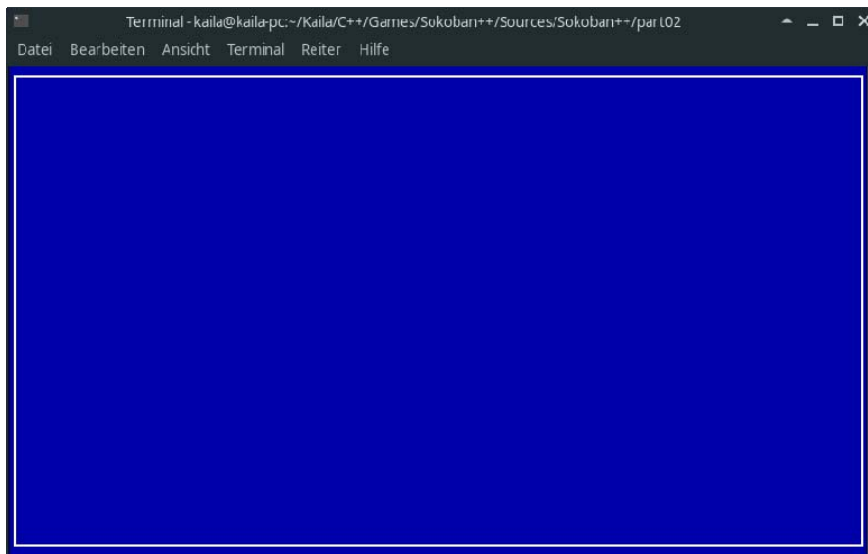


Abbildung 11: Das farbige Fenster als Spielfläche

5.4.D Triggerpunkt Part 02

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.5 Part 03 – Ein beliebiges Spielfeld

Das Sokoban-Spielfeld wird in einem extra Modul implementiert. Es besteht aus `gameboard.cpp` und `gameboard.hpp`. Das Spielfeld enthält eine Klassendefinition mit allen relevanten Eigenschaften und Methoden. Klassendefinition? Eigenschaften und Methoden? Was ist das?

5.5.A Exkurs – Objektorientierte Programmierung

Objektorientiertes Programmieren (OOP) ist ein Programmierparadigma basierend auf Objekte (mit **Eigenschaften** und **Methoden**), um die Vorteile der Modularisierung und Wiederverwendbarkeit auszunutzen. Es werden **Objekte** verwendet, die in der Regel Instanzen von **Klassen** sind. Diese Objekte besitzen Eigenschaften und interagieren untereinander durch ihre Methoden, um Anwendungen und Computerprogramme zu realisieren.

5.5.B Part 03-1 – Das Gameboard-Modul via UML

Das UML-Klassendiagramm des Gameboard-Moduls ist nachfolgend gezeigt:

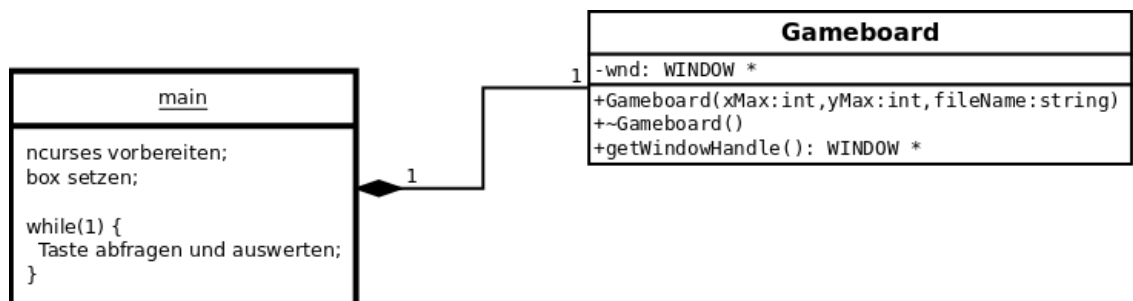


Abbildung 12: Klassendiagramm Part 03-1

UML? Was ist das nun schon wieder?

UML (Unified Modeling Language) ist ein Standard der **OMG** (<http://www.omg.org/uml>) und definiert eine Notation und Semantik zur Visualisierung, Konstruktion und Dokumentation von Modellen für die objektorientierte Softwareentwicklung – also für die objektorientierte Programmierung. Das Klassendiagramm ist eines der wichtigsten Diagrammtypen in UML und wird im weiteren Verlauf als Modulbeschreibung herangezogen.

Wie das Klassendiagramm interpretiert wird, beschreibt nachfolgender Exkurs.

5.5.B.1 Exkurs: Das Klassendiagramm

Der rechte Kasten (Gameboard) in Abbildung 12 besteht aus 3 Bereiche. Er dient lediglich als „**Bauplan**“, wie ein Objekt erstellt werden soll:

1. **Name** der Klasse (Gameboard)
2. **Eigenschaften** der Klasse (`wnd`)
3. **Methoden** der Klasse (`Gameboard(...)`, `~Gameboard()`, `getWindowHandle()`)

5.5.B.1.1 Name der Klasse

Der Name der Klasse kann frei gewählt werden. I.d.R. werden Namen groß angefangen und ggf. aneinandergereiht.

5.5.B.1.2 Eigenschaften der Klasse

Die Eigenschaften beschreiben **statische Zustände** einer Klasse. Die Farbe der Haare wäre eine Eigenschaft der Klasse Mensch oder in unserem Beispiel das Handle `wnd` des Gameboard-Spielfelds. Eigenschaften von Klassen besitzen aber noch besondere Sichtbarkeiten, die ihnen vorangestellt sind.

5.5.B.1.3 Methoden der Klasse

Methoden beschreiben **dynamische Prozesse**, die bei Aufruf ausgeführt werden. Sie ähneln C-Funktionen, können aber überladen oder überschrieben werden. Zwei Methoden müssen mit wenigen Ausnahmen implementiert werden:

- Einen Konstruktor: Er wird beim Erstellen eines Objekts der Klasse aufgerufen. Er hat keinen Rückgabewert und demzufolge auch keinen Rückgabotyp.
- Einen Destruktor: Er wird beim Löschen eines Objekts der Klasse aufgerufen. Er hat ebenso keinen Rückgabewert und demzufolge auch keinen Rückgabotyp. Vor dem Methodennamen muss ein ‘~’ vorangestellt werden, um den Destruktor vom Konstruktor zu unterscheiden.

5.5.B.1.4 Sichtbarkeiten

Für Eigenschaften und Methoden müssen Sichtbarkeiten angegeben werden. Folgende Sichtbarkeiten sind möglich:

- ‘-‘ bedeutet Private: Eigenschaften oder Methoden können nur innerhalb der Klasse genutzt werden
- ‘+‘ bedeutet Public: Eigenschaften oder Methoden können überall, also über die Klasse hinaus genutzt werden
- ‘◇‘ bedeutet Protected: Eigenschaften oder Methoden können von abgeleiteten Klassen genutzt werden

- **Eigenschaften bitte immer Private!** Über Getter- und Setter-Methoden können Eigenschaften von außen beeinflusst werden.
- Viele Methoden sind Public! Nur interne Methoden können Private deklariert werden.
- Die Sichtbarkeit Protected ist fast nie nötig!

5.5.B.1.5 Assoziationen zwischen Klassen

Es existieren einige mögliche Assoziationen, die zwischen Klassen implementiert werden können. In Abbildung 12 ist eine **starke Aggregation** gewählt worden, eine sog. **Komposition**:

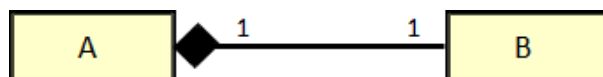


Abbildung 13: UML-Komposition

Das Objekt der Klasse A erzeugt ein Objekt der Klasse B. Das kann im Konstruktor der Klasse A erledigt werden. Darüber hinaus werden häufig die sog. *Multiplizitäten* angegeben. Das sind die Zahlen an den jeweiligen Endpunkten (hier beide 1). Das sagt aus, dass das Objekt der Klasse A genau ein Objekt der Klasse B erzeugt. Öfters wird auch ein Bereich verwendet (Bsp.: 1..*). Es besagt, dass beliebig viele Objekte der Klasse B erzeugt werden, aber mindestens 1. Das wird dann mit einem `Vector` oder einer `List` implementiert.

5.5.B.2 Die Implementierung

Es folgt nun die Implementierung des Klassendiagramms in Abbildung 12. Kopieren Sie ~/part02 nach ~/part03. Zuerst werden in ~/part03 gameboard.hpp und gameboard.cpp erzeugt.

Der Inhalt von gameboard.hpp beinhaltet die **Klassendefinitionen** inkl. **Eigenschaften** und **Methoden-Prototypen**. Die Klassendefinition ist der Bauplan der daraus abgeleiteten Objekte, die bisher noch nicht erstellt sind:

```

1 #ifndef GAMEBOARD_HPP // Sentinel (Wächter), verhindert das
2 #define GAMEBOARD_HPP // Mehrfachladen der Headerdatei!
3
4 #include <string> // Verweis auf string
5 #include <ncurses.h> // Verweis auf ncurses
6
7 using namespace std; // sollte klar sein!
8
9 class Gameboard { // Klassendefinition
10 private: // Sichtbarkeit
11     WINDOW *wnd; // Eigenschaft
12
13 public: // Sichtbarkeit
14     Gameboard(int, int, string); // Konstruktor
15     virtual ~Gameboard(); // Destruktor
16     virtual WINDOW *getWindowHandle(); // eine Methode
17 };
18
19 #endif

```

Zeile	Funktion
1,2,19	Der Sentinel (Wächter) sorgt dafür, dass der Inhalt der Headerdatei nur einmal eingefügt wird. „ Der erste gewinnt! “. Problem wäre überlagerte Eigenschaften und Methoden.
4,5	Verweise auf die string- und ncurses-Bibliothek
7	Namespace...
9,17	Klassendefinition mit beliebigen Namen, bitte sinnvoll.
10,11	Deklaration der Eigenschaften mit der entsprechenden Sichtbarkeit
13-16	Deklaration der Methoden mit der entsprechenden Sichtbarkeit. Es müssen mindestens 2 Methoden definiert werden. <ul style="list-style-type: none"> • Konstruktor: Aufruf beim Erstellen eines Objekts der Klasse • Destruktor: Aufruf beim Löschen eines Objekts der Klasse Ansonsten können beliebig viele Methoden mit beliebig vielen Sichtbarkeiten definiert werden.

Der Inhalt von gameboard.cpp beinhaltet die Implementierung der Methoden, die in der Klassendefinition spezifiziert sind. Erstellte Objekte sind damit in der

Lage, Aktivitäten auszuführen. Nach der Implementierung sind die Objekte immer noch nicht erstellt:

```

1 #include <iostream>
2 #include <ncurses.h>
3 #include <string>          // Standard Header
4
5 #include "gameboard.hpp"  // Einbinden der eigenen Modul-Headerdatei
6
7 using namespace std;     // no Comment ©
8
9 Gameboard::Gameboard(int xMax, int yMax, string fileName) {
10     initscr();           // Ncurses: Initialisierung + neues Fenster
11     wnd = newwin(yMax, xMax, 1, 1);
12     cbreak();
13     noecho();
14     clear();
15
16     keypad(wnd, true);   // schaltet Sondertasten ein
17     curs_set(0);        // Cursor nicht sichtbar
18     start_color();      // Farben einschalten
19
20     init_pair(1, COLOR_WHITE, COLOR_BLUE);
21     wbkgd(wnd, COLOR_PAIR(1)); // Hintergrund/Vordergrundfarben
22
23     attron(A_BOLD);     // Box zeichnen
24     box(wnd, 0, 0);
25     attroff(A_BOLD);
26 }
27
28 Gameboard::~Gameboard() {
29 }
30
31 WINDOW *Gameboard::getWindowHandle() {
32     return wnd;
33 }

```

Zeile	Funktion
1-3	Headerdateien..
5	Einbinden der eigenen Modul-Headerdatei. Immer notwendig..
9	Definifion des Konstruktors der Form: Klasse::Konstruktor. Es können beliebig viele Parameter übergeben werden, die i.d.R. Klasseigenschaften zugeordnet werden. In unserem Fall nicht!
10-25	Auslagerung der Ncurses-Funktionen in den Konstruktor der Klasse Gameboard. Wenn ein Objekt der Klasse erstellt wird, wird automatisch der Konstruktor und entsprechend die Ncurses-Funktionen aufgerufen. Vorteil: Der Nutzer braucht sich nicht um Ncurses zu kümmern!
28-29	Definifion des Destruktors der Form: Klasse::~Destruktor. Er ist hier im Moment leer. Er kann sinnvoll sein, wenn beim Löschen des Objekts „ Aufräumarbeiten “ erledigt werden müssen!
31-33	Definifion einer Klassenmethode der Form: Klasse::Name. Auch hier bitte sinnvolle Namen. getWindowHandle sagt eigentlich alles aus.

Wichtig ist hier der Rückgabewert. WINDOW * gibt den Ncurses-Fensterhandler zurück. Der * steht für eine Instanz auf das Fenster, also ein Verweis!

Nun ist das Modul Gameboard als Klasse – **Bauplan** – definiert. Es können jetzt beliebig viele **Objekte** der **Klasse** erstellt werden, aber bitte das Klassendiagramm in Abbildung 12 beachten: **Nur ein Objekt!**

Das Hauptprogramm main.cpp erstellt oder instanziiert ein Objekt (Zeile 16):

```
1 #include <string>
2 #include <memory>
3 #include <unistd.h>
4 #include <ncurses.h>
5
6 #include "gameboard.hpp"
7
8 using namespace std;
9
10 #define WAIT_TICK 10000 // Konstante, nicht veränderbar
11
12 WINDOW *wnd;
13 shared_ptr<Gameboard> gb; // Smart-Pointer auf das Gameboard-Objekt
14
15 int init() {
16     gb = make_shared<Gameboard>(10, 10, ""); // Erstellen eines Gameboard-Objekts
17     wnd = gb->getWindowHandle(); // Holen des Ncurses-Handlers
18     return 0; // Rückgabe immer 0
19 }
20
21 void run() {
22
23     int in_char;
24     bool exit_requested = false;
25
26     while(!exit_requested) {
27         in_char = wgetch(wnd);
28         switch(in_char) {
29             case 'q':
30                 exit_requested = true;
31                 break;
32             case KEY_UP:
33             case 'w':
34                 break;
35             case KEY_DOWN:
36             case 's':
37                 break;
38             case KEY_LEFT:
39             case 'a':
40                 break;
41             case KEY_RIGHT:
42             case 'd':
43                 break;
44             default:
45                 break;
46         }
47         usleep(WAIT_TICK);
48         wrefresh(wnd);
49     }
50 }
51
52 void close() {
53     endwin();
54 }
55
56 int main () {
```

```

57  init();
58  run();
59  close();
60  }

```

Zeile	Funktion
10	Eine Konstante mit dem Namen <code>WAIT_TICK</code> und dem Wert <code>10000</code> wird definiert. Das ist eine <i>waschechte</i> Prä-Prozessor -Anweisung.
13	Ein Smart-Pointer auf ein Gameboard-Objekt wird erzeugt. In diesem Fall ist es ein <code>shared_ptr</code> , also ein Zeiger auf ein Objekt, das von mehreren anderen Objekten genutzt werden kann (shared)
16	Ein Gameboard-Objekt wird mit <code>make_shared</code> erzeugt. In den Klammern <code><></code> muss ein Objekt oder eine primitiver Datentyp angegeben werden. Wird ein Objekt angegeben, muss in den Klammern <code>()</code> die Parameterliste des Objekt-Konstruktors eingetragen werden. Der Gameboard-Konstruktor erwartet 3 Parameter: <code>xMax</code> , <code>yMax</code> und <code>fileName</code> . Das Fenster hat eine Fläche von <code>10x10</code> Pixel.
17	Holen des Ncurses-Handlers über die sog. Getter-Methode <code>getWindowHandle</code> . Der Handler wird für die Tastatursteuerung und für Input/Output-Aktivitäten benötigt.

Damit das Gameboard-Modul in den Kompilierungsvorgang eingebunden wird, muss folgendes in dem `Makefile` ergänzt werden:

```
5 OBJS = main.o gameboard.o
```

Bis hierhin ist der Quellcode detailliert dargestellt. Sie sollen nicht nur das **Kopieren**, sondern auch das **selbstständige Planen und Implementieren** von Software-Modulen erlernen. Deshalb wird zunehmend auf detailreiche Implementierungen verzichtet. Die Klassendiagramme sind dagegen vollständig und müssen komplett umgesetzt werden!

5.5.B.3 Triggerpunkt Part 03-1

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.5.C Part 03-2 – Das Pixel als Zusammenschluss

Das UML-Klassendiagramm des Gameboard-Moduls wird folgendermaßen erweitert:

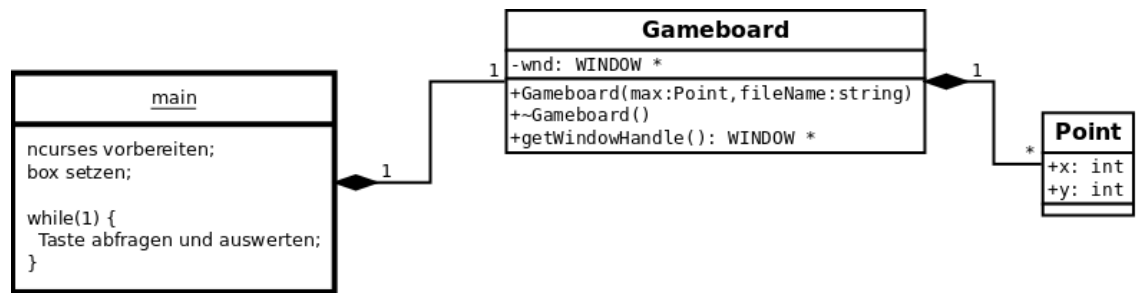


Abbildung 14: erweitertes Klassendiagramm Part 03-2

Die Klasse `Point` ist gegenüber dem vorherigen Modell hinzugekommen. Die Klasse `Gameboard` kann beliebig oft ein Objekt der Klasse `Point` erzeugen. Die Klassendarstellung verrät noch, dass es keine Methoden gibt. Die Eigenschaften `x` und `y` sind `Public`.

Erinnerung: Eigenschaften niemals Public! Mit einer Ausnahme: Die `Point`-Klasse wird nicht als herkömmliche Klasse verwendet, sondern als Struktur, sozusagen als Variablenzusammenschluss.

Die Klasse `Point` muss der `gameboard.hpp` zugefügt werden:

```

13 class Point { // Point-Klasse als Struktur-Ersatz
14     public: // public-Attribute hier sinnvoll
15         int x;
16         int y;
17 };
  
```

5.5.C.1 Anlegen eines Point-Objekts

Ein `Point`-Objekt wird im Speicher so angelegt:

```
Point max;
```

5.5.C.2 Schreiben der Point-Komponenten

Auf die Komponenten wird mit der Punkt-Notation zugegriffen:

```
max.x = 7;
```

```
max.y = 5;
```

5.5.C.3 Lesen der Point-Komponenten

Auch hier gilt die Punkt-Notation:

```
int x = max.x;
```

```
int y = max.y;
```

Den Konstrktor der Gameboard-Klasse passen Sie selbstständig an!

5.5.D Das Resultat



Abbildung 15: Das Spielfeld 10x10 Pixel

5.5.E Triggerpunkt Part 03

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.6 Part 04 – Das Sokoban-Spielfeld

Für Sokoban wird ein 2-dimensionales Spielfeld benötigt. Der Aufbau ist in Abbildung 9 zu sehen. Mit Hilfe der Ausgabefunktionen der Ncurses können via X/Y-Koordinaten Ausgaben und Abfragen getätigt werden.

Das UML-Klassendiagramm des Gameboard-Moduls wird folgendermaßen erweitert:

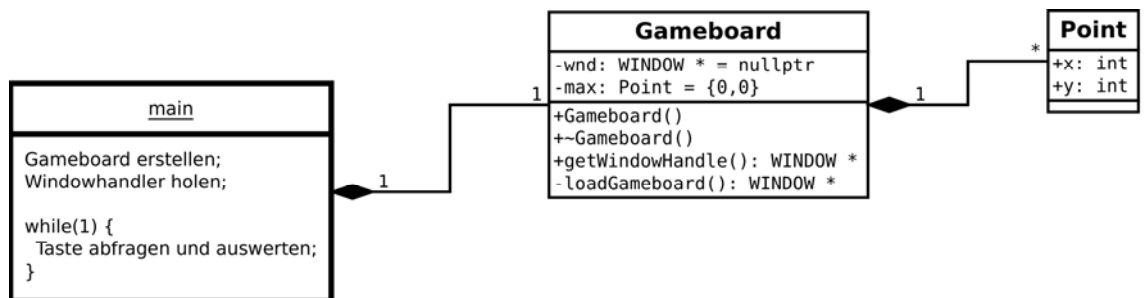


Abbildung 16: erweitertes Klassendiagramm Part 04

Wie man sieht, ist die Methode `loadGameboard` dazugekommen, die auch noch die Sichtbarkeit `Private` besitzt. Sie wird nur innerhalb der Klasse benutzt. In der Methode soll folgende Aufgabe ausgegeben werden:

	0	1	2	3	4	5	6	7	8	9
0	#	#	#	#	#	#	#	#		
1	#							#		
2	#							#		
3	#							#	#	#
4	#									#
5	#							#	#	#
6	#	#	#	#	#	#	#	#		

Abbildung 17: Testaufgabe mit Spielfeldumrandung

5.6.A Die Ncurses-Ausgabe der Aufgabe

Für die Ausgabe der Aufgabe in Abbildung 17 ist folgendes Codefragment nützlich:

```

32 WINDOW *Gameboard::loadGameboard() {
33     WINDOW *w;                // Fenster-Handler
34     char ch = '#';           // Zeichen deklarieren
35
36     max = {10, 7};           // maximale Spielfeld (x/y)
37     w = newwin(max.y, max.x, 1, 1); // neues Fenster erzeugen
38
39     mvwaddch(w, 0, 0, ch);    // Zeichen ins Fenster mit den
40     mvwaddch(w, 0, 1, ch);    // entsprechenden Koordinaten
41     ...                       // zeichnen
116    return w;                // Fenster-Handler zurückgeben
117 }

```

Zeile	Funktion
32	Der Methoden-Kopf <code>loadGameboard</code>
33	Eine Hilfsvariable <code>w</code> vom Datentyp <code>WINDOW *</code>
34	Eine Variable <code>ch</code> vom Datentyp <code>char</code> wird deklariert und mit dem Zeichen <code>'#'</code> initialisiert.
36	Die Klasseneigenschaft <code>max</code> wird mit Testdaten belegt: <code>x, y</code>
37	Ein neues Ncurses-Fenster wird angelegt, in dem das Spielfeld gezeichnet werden soll. Es bezieht sich auf die Eigenschaft <code>max</code> und den Handler <code>w</code>
39...	Die Funktion <code>mvwaddch</code> (move window add character) zeichnet innerhalb eines Fensters <code>w</code> mit den Koordinaten <code>y/x</code> das Zeichen <code>ch</code>
116	Der Fenster-Handler <code>w</code> wird dem Aufrufer zurückgegeben

Implementieren Sie die Methode `loadGameboard`, die die Testaufgabe zeichnet. Rufen sie die private Methode `loadGameboard` im **Konstruktor** auf und speichern Sie den zurückgegebenen Handler.

5.6.B Der Datentyp `char`

Der Datentyp `char` (Character) ist in der Lage, ganzzahlige Werte zu speichern. Nach Standard ist er genau 1 Byte breit. Wird der Datentyp als solcher verwendet, ist er `signed`, also vorzeichenbehaftet. Stellt man das Schlüsselwort `unsigned` dafür, ist er vorzeichenlos.

Bits	Format	Wertebereich
8	(signed) char	-128 to +127
	unsigned char	0 to 255

5.6.C Die ASCII-Tabelle

Der Datentyp `char` wird häufig für Zeichendarstellungen verwendet. Die ASCII-Tabelle (Standard-Codes 7-Bit-Zeichenkodierung) enthält alle Standard-Codes der in der Konsole darstellbaren Zeichen. Ein Ausschnitt der Tabelle ist hier gezeigt:

DEZ	HEX	OKT	HTML	MSB	LSB	Bezeichnung
32	20	040	 	P010	0000	SP (Leerzeichen)
33	21	041	!	P010	0001	!
34	22	042	"	P010	0010	"
35	23	043	#	P010	0011	#
36	24	044	$	P010	0100	\$
37	25	045	%	P010	0101	%
38	26	046	&	P010	0110	&
39	27	047	'	P010	0111	>
40	28	050	(P010	1000	(
41	29	051)	P010	1001)
42	2A	052	*	P010	1010	*
43	2B	053	+	P010	1011	+
44	2C	054	,	P010	1100	,
45	2D	055	-	P010	1101	-
46	2E	056	.	P010	1110	.
47	2F	057	/	P010	1111	/

Abbildung 18: Ausschnitt der ASCII-Tabelle mit dem #-Zeichen

In Zeile 34 (`char ch = '#'`) wird die Variable `ch` mit dem Zeichen `'#'` initialisiert. Wie aus der Tabelle zu sehen ist, ist dem Zeichen `'#'` der Wert 35_{10} oder 23_{16} zugeordnet. Genau das steht in der Variablen `ch`.

Zeichen werden **grundsätzlich** über die ASCII-Tabelle darstellbar. Variablen enthalten nur die zugewiesenen **Werte!**

Andere Möglichkeiten:

<code>char ch = 35;</code>	Zeichen <code>'#'</code> als Dezimalwert
<code>char ch = 0x23;</code>	Zeichen <code>'#'</code> als Hexadezimalwert

5.6.D Das Resultat



Abbildung 19: Das Spielfeld

5.6.E Die Zahlensysteme

5.6.E.1 Das binäre Zahlensystem

Das Binärsystem oder auch Dualsystem kennt nur zwei verschiedene Ziffern für die Darstellung von Zahlen: **0** und **1**. Die Basis des Zahlensystems ist 2, die Wertigkeit wird im Exponenten dargestellt. Eine Binärzahl **1101 0010₂** ist folgendermaßen codiert und kann entsprechend in das Dezimalsystem umgewandelt werden:

1	1	0	1	0	0	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	0	16	0	0	2	0
Summe						210₁₀	

Eine Dezimalzahl **210₁₀** kann nach dem Resteverfahren in eine Binärzahl gewandelt werden. Dazu wird die Dezimalzahl jeweils mit der Basis (2) Integerdividiert und der Rest gebildet:

$$210 : 2 = 105, \text{ Rest } 0$$

$$105 : 2 = 52, \text{ Rest } 1$$

$$52 : 2 = 26, \text{ Rest } 0$$

$$26 : 2 = 13, \text{ Rest } 0$$

$$13 : 2 = 6, \text{ Rest } 1$$

$$6 : 2 = 3 \quad , \text{ Rest } 0$$

$$3 : 2 = 1 \quad , \text{ Rest } 1$$

$$1 : 2 = 0 \quad , \text{ Rest } 1 \quad \rightarrow \text{ Bei Ergebnis } 0 \text{ ist Schluss!}$$

Die kleinste Wertigkeit ist oben, die größte unten. Die Binärzahl wird deshalb von unten nach oben links beginnend aufgeschrieben: **1101 0010₂**

5.6.E.2 Das hexadezimale Zahlensystem

Das Hexadezimalsystem hat die Basis 16 und besteht somit aus 16 verschiedenen Ziffern. Mit den Dezimalzahlen 0 bis 9 können bereits 10 Ziffern dargestellt werden, für die restlichen 6 Ziffern werden die ersten Buchstaben des Alphabets verwendet, also **0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f**. Eine Hexadezimalzahl **3b₁₆** (0x3b) ist folgendermaßen codiert und kann entsprechend in das Dezimalsystem umgewandelt werden:

3	b
3·16¹	11·16⁰
Summe	59₁₀

Eine Dezimalzahl **59₁₀** kann nach dem Resteverfahren in eine Hexadezimalzahl gewandelt werden. Dazu wird die Dezimalzahl jeweils mit der Basis (16) Integer-dividiert und der Rest gebildet:

$$59 : 16 = 3, \text{ Rest } 11 \text{ (b)}$$

$$3 : 16 = 0, \text{ Rest } 3 \quad \rightarrow \text{ Bei Ergebnis } 0 \text{ ist Schluss!}$$

Die kleinste Wertigkeit ist oben, die größte unten. Die Hexzahl wird deshalb von unten nach oben links beginnend aufgeschrieben: **3b₁₆**

5.6.E.3 Umrechnung Bin in Hex und Hex in Bin

Die Umrechnung einer Binärzahl in eine Hexadezimalzahl oder umgekehrt ist besonders einfach. **4 Stellen** einer Binärzahl entspricht genau **eine** hexadezimale **Stelle**. Wird die Binärzahl **1101 0011₂** in eine Hexadezimalzahl umgewandelt, werden immer 4 Bits von rechts nach links betrachtet:

1101	0011
D (13)	3

Nicht vorhandene Bits können von links nach rechts mit **0** aufgefüllt werden.
Folgende Binärzahl soll umgewandelt werden **10 1101**₂.

0010	1101
2	D (13)

5.6.F Triggerpunkt Part 04

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.7 Part 05 – Ein Spieler kommt ins Spiel

Das bisherige Spielfeld ist leer und beinhaltet weder Spieler noch Kisten oder Ziele. Ein Spieler wird nun benötigt, um zukünftige Aufgaben zu lösen. Er wird mit dem Zeichen '@' dargestellt.

5.7.A Part 05-1 – Ein Spieler betritt das Spielfeld

Der Spieler wird an entsprechender Stelle im Spielfeld positioniert. Er kann sich noch nicht bewegen. Er bleibt an dieser Stelle, zum Beispiel hier:

	0	1	2	3	4	5	6	7	8	9
0	#	#	#	#	#	#	#	#		
1	#							#		
2	#							#		
3	#							#	#	#
4	#								@	#
5	#							#	#	#
6	#	#	#	#	#	#	#	#		

Abbildung 20: Testaufgabe mit Spieler

Das UML-Klassendiagramm muss entsprechend erweitert werden:

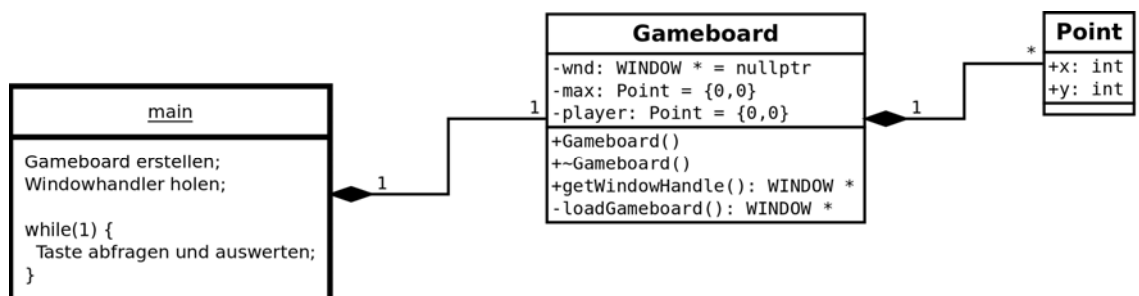


Abbildung 21: erweitertes Klassendiagramm Part 05-1

Die Implementierung ist entsprechend einfach. Sie benötigen die X/Y-Koordinaten der Position.

5.7.A.1 Das Resultat

Nach dem Kompilieren und starten des Spiels muss folgendes Bild entstehen:

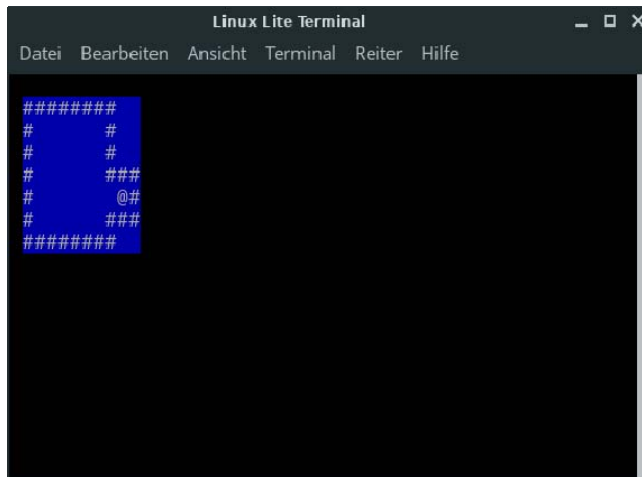


Abbildung 22: Der Spieler auf dem Spielfeld

5.7.B Part 05-2 – Der Spieler bewegt sich

Der Spieler muss sich innerhalb des Spielfelds bewegen. Deshalb müssen die entsprechenden Tasten, die in 5.4.A eingeführt wurden, mit Leben gefüllt werden. Schaut man sich das Koordinatensystem in Abbildung 9 an, wird schnell klar, wie die Bewegung in `main.cpp` implementiert werden muss.

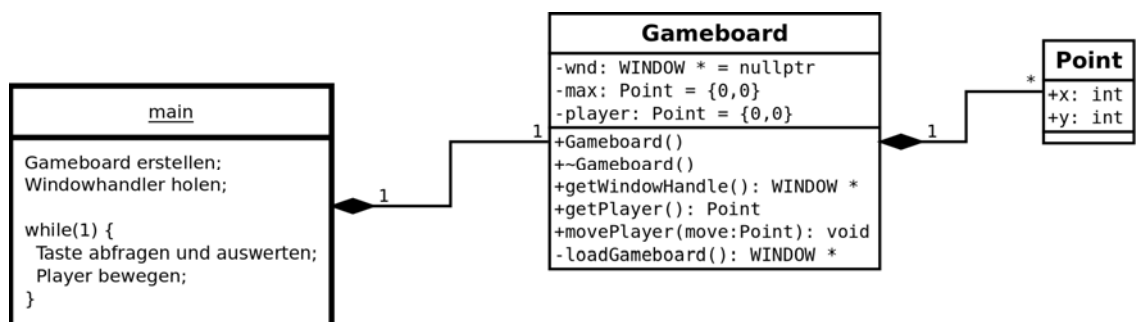


Abbildung 23: erweitertes Klassendiagramm Part 05-2

```

28 Point move;
35 move = gb->getPlayer();
45 move.y -= 1; // Spieler ein Schritt nach Norden (w)
46 gb->movePlayer(move); // Spieler gewegt sich dort hin
  
```

Zeile	Funktion
28	Deklaration einer lokalen Variablen <code>move</code> in der Funktion <code>run()</code>
35	Die neue Methode <code>getPlayer()</code> holt die aktuelle Position des Spielers und gibt die Koordinaten mittels <code>Point</code> zurück
45	Die aktuelle Spielerposition (<code>move</code>) wird entsprechend angepasst. Das Koordinatensystem gibt Auskunft über die Werte!
46	Der Spieler führt die Bewegung mittels der neuen Methode <code>movePlayer(pos)</code> durch. <u>Vorsicht:</u> Der Spieler muss beim Verlassen der Position „restauriert“ werden. <u>Tipp:</u> <code>mvwaddch()</code> , <code>wrefresh()</code>

5.7.B.1 Das Resultat



Abbildung 24: Der Spieler bewegt sich (aber noch ohne Kollisionsabfrage)

5.7.B.2 Die Spielerrestauration

Wenn der Spieler bewegt wird, muss an seinem vorherigen Platz der Inhalt wiederhergestellt werden. Folgende Abbildung zeigt das Problem:

	0	1	2	3	4	5	6	7	8	9
0	#	#	#	#	#	#	#	#		
1	#							#		
2	#							#		
3	#			↑ @				#	#	#
4	#									#
5	#							#	#	#
6	#	#	#	#	#	#	#	#		

Für jeden Schritt:
 1.) zukünftige Position merken
 2.) Spieler auf Position setzen
 3.) vorherige Position restaurieren

Am Beispiel Nord:
 1.) Inhalt 2,3 merken
 2.) Spieler auf Position 2,3 setzen
 3.) vorherigen Inhalt auf 3,3 setzen

Vereinfacht Nord:
 1.) Spieler auf Position 2,3 setzen
 2.) Leerzeichen auf Position 3,3 setzen

Abbildung 25: Restauration von Inhalten bei Bewegungen

In unserem Spiel wird lediglich die vereinfachte Form der Restauration genutzt. Später wird eine andere Möglichkeit der Standard-Restauration vorgestellt.

5.7.C Part 05-3 – Der Spieler und die Kollisionsabfrage

Im vorherigen Abschnitt bewegt sich der Spieler durch das Spielfeld, aber wird durch die Wand `#` nicht begrenzt. Es fehlt die Kollisionsabfrage. Das UML-Diagramm ist wie in 5.7.B, es muss lediglich nur die Methode `movePlayer()` erweitert werden. Mit der Ncurses-Funktion `mvwinch()` können einzelne Zeichen eines Fensters gelesen werden. Für eine Kollisionsabfrage muss folgendes durchgeführt werden:

1. Zukünftiges Zeichen lesen und prüfen, ob es eine Wand `#` ist
2. Ist es eine Wand, dann keine Bewegung des Spielers zulassen
3. Ist es keine Wand, Bewegung des Spielers durchführen

```

47 char ch = mvwinch(wnd, y, x) & A_CHARTEXT;
48 if('#' == ch) {
49     // Wand vorhanden
50 } else {
51     // Wand nicht vorhanden
52 }

```

Zeile	Funktion
47	Die Ncurses-Funktion <code>mvwinch()</code> gibt ein Zeichen eines Fensters bei einer X/Y-Koordinate zurück. <code>A_CHARTEXT</code> ist eine Bitmaske und sorgt dafür, dass nur Zeichen zurückgegeben werden.
48-52	Auswertung des zurückgegebenen Zeichens

2. Debugger `gdb` mit dem ausführbaren Programm starten
3. Breakpoint setzen mit `break <zeilennummer>`
4. Programm im Debugger laufen lassen mit `run`. Programm stoppt bei der Zeilennummer, die mit `break` angegeben wurde.
5. Analyse von Variablen mit `print <Variablenname>` oder `display <Variablenname>`
6. Programm schrittweise fortlaufen lassen mit `next`. Methoden werden übersprungen.
7. Programm schrittweise fortlaufen lassen mit `step`. Der Debugger springt in Methoden hinein, um dort weiterzulaufen.
8. Programm dauernd bis zum nächsten Breakpoint laufen lassen mit `cont`
9. Programm und Debugger beenden mit `quit`

Eine detaillierte Beschreibung der Befehle von `gdb` ist auf der Homepage zu finden. Ein Ncurses-Frontend `gdbtui` für den `gdb` sorgt für eine vereinfachte Bedienung während der Fehlersuche. `gdbtui` ist Bestandteil der Standard-`gdb`-Installation. Im Anhang befindet sich eine GDB-Referenzkarte: 8.3.

5.7.C.4 Triggerpunkt Part 05

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.8 Part 06 – Kisten und Ziele positionieren

In diesem Abschnitt werden Kisten und Ziele auf dem Spielfeld positioniert. Weiterhin muss die Restauration nach Abbildung 25 auf Standard (nicht einfach) geändert werden. Die Kisten und Ziele werden nach folgender Abbildung eingebaut:

	0	1	2	3	4	5	6	7	8	9
0	#	#	#	#	#	#	#	#		
1	#							#		
2	#		\$		\$			#		
3	#							#	#	#
4	#	#	#	#					@	#
5	#	.	.					#	#	#
6	#	#	#	#	#	#	#	#		

Abbildung 27: Komplette Testaufgabe mit Kisten und Ziele

Anschließend muss die Restauration des Hintergrunds modifiziert werden. Folgendes UML-Diagramm gibt darüber Auskunft:

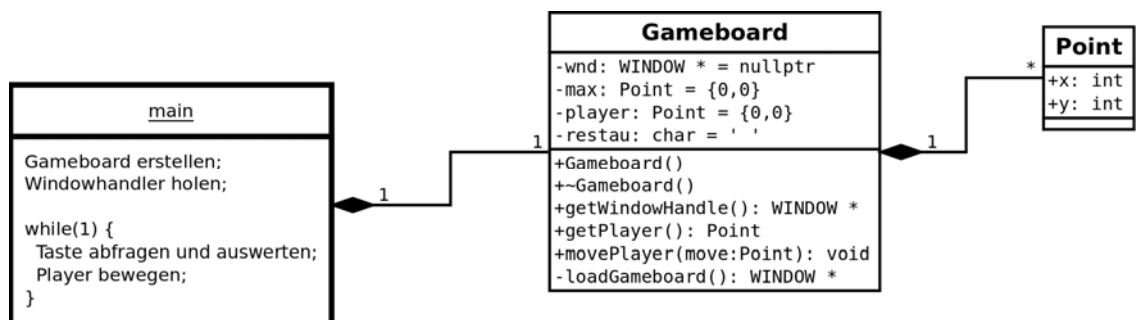


Abbildung 28: erweitertes Klassendiagramm Part 06

Benötigt wird die Eigenschaft `restau` für das Zwischenspeichern des Inhalts. Implementiert wird die Restauration nach Abbildung 25: **Am Beispiel Nord**.

5.8.A Das Ergebnis



Abbildung 29: Spieler läuft über Kisten mit Restauration

5.8.B Triggerpunkt Part 06

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.9 Part 07 – Die Kisten verschieben

Der Spieler muss alle Kisten auf die Ziele bewegen, um die Aufgabe zu lösen. Die Spielbedingungen sind in 4.1 beschrieben. Nun ans Werk, um das Kisten-schieben zu implementieren.

5.9.A Part 07-1 – Das Logger-Modul

In 5.7.C.3 ist der Debugger `gdb` vorgestellt worden. Eine einfachere Methode ist es, Debug-Informationen in eine Datei zu schreiben. Dazu dient das Logger-Modul `logger.cpp` und `logger.hpp`. Das Modul wird zur Verfügung gestellt.

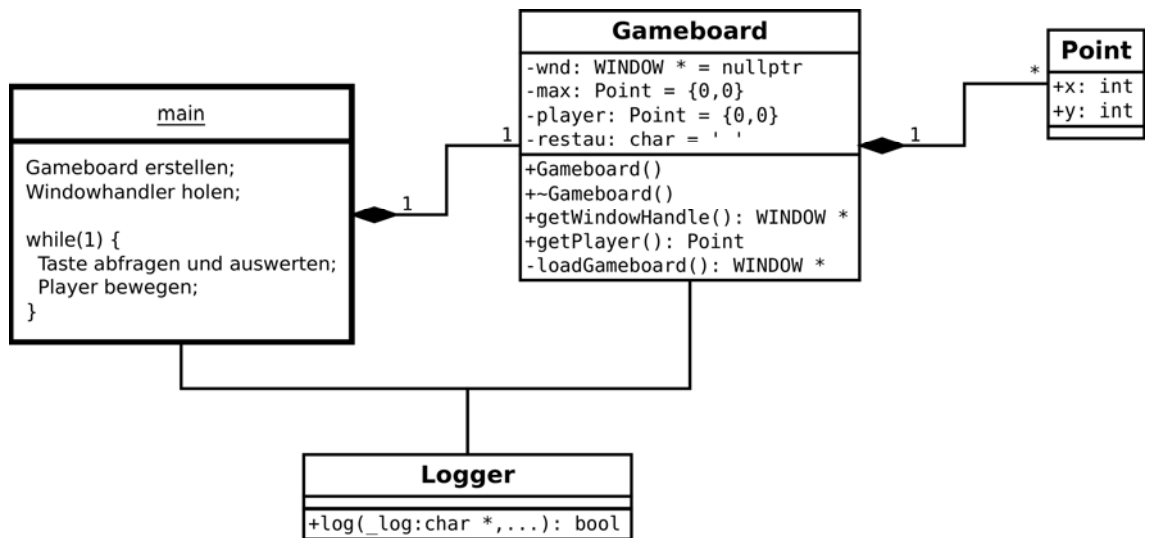


Abbildung 30: erweitertes Klassendiagramm Part 07-1

Das Anlegen einer Debug-Information in `log.txt` wird folgendermaßen initiiert:

```
114 Logger::getInstance()->log(
115     (char *)"TEST: x=%i/y=%i",
116     x, y);
```

Zeile	Funktion
114	Aus der Logger-Klasse (<code>Logger:: getInstance()</code>) wird die Methode <code>log()</code> aufgerufen. Dieser Aufbau ist immer gleich.
115	Formatierungsstring mit ggf. Text und beliebige Variablenformatierungen (<code>%i</code> : Integer-Variablen, hier 2 Stück)
116	Beliebige Anzahl der zu analysierenden Variablen (x und y, 2 Stück)

Ist $x=10$ und $y=3$, so wird in `log.txt` folgendes stehen:
`logger.cpp (Line 40): TEST: x=10/y=3`

Es können Variablen mit unterschiedlichen Datentypen dargestellt werden. Anbei die wichtigsten.

Formatierung	Variablen
%i o. %d	Integer-Variablen
%c	Zeichen-Variablen
%s	String-Variablen
%f	Float-Variablen

Für alle möglichen Formatierungen gibt die `printf`-Manpage Auskunft.

5.9.B Part 07-2 – Kiste für Kiste schieben

Nun ist es Zeit, Kisten zu verschieben. Um das Problem zu verdeutlichen, ist folgende Abbildung zu betrachten:

	0	1	2	3	4	5	6	7	8	9
0	#	#	#	#	#	#	#	#		
1	#							#		
2	#		\$		\$			#		
3	#				@			#	#	#
4	#	#	#	#	\$					#
5	#	.	.				.	#	#	#
6	#	#	#	#	#	#	#	#		

Abbildung 31: Verschieben einer Kiste

Der Spieler möchte eine Kiste nach Norden verschieben (siehe Abbildung 31). Folgender Ablauf ist sinnvoll:

- Prüfen, ob Spieler vor eine Kiste steht: Ja
- Prüfen, ob Kiste vor einer anderen oder vor einer Wand steht: Nein
- Kiste von (2,4) nach (1,4) verschieben
- Spieler von (3,4) nach (2,4) verschieben
- Vorherige Spielerposition (3,4) restaurieren

Vorherige Kistenposition muss NOCH nicht restauriert werden, da der Spieler dorthin verschoben wird!

Das zugehörige UML-Klassendiagramm sieht so aus:

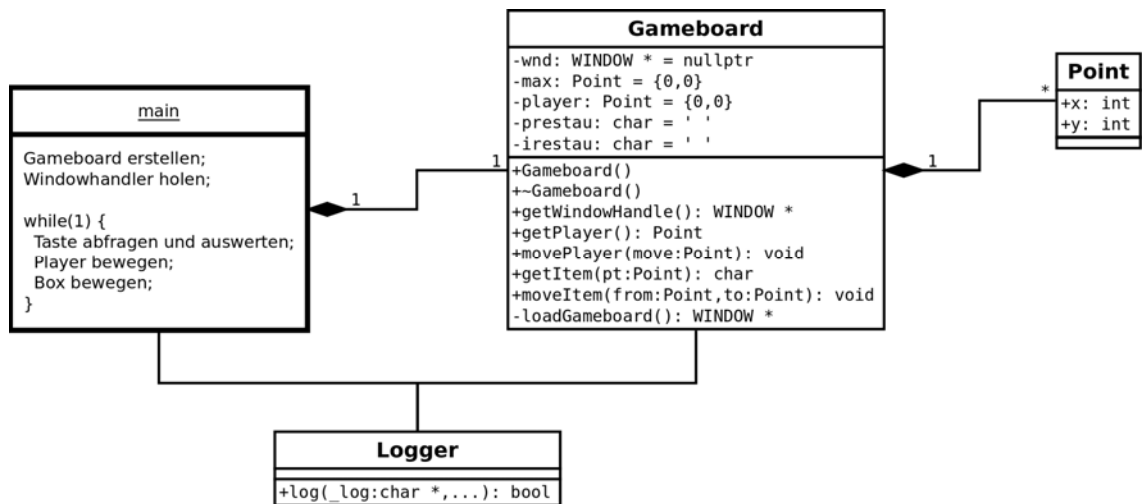


Abbildung 32: erweitertes Klassendiagramm Part 07-2

Methoden	Funktion
getItem()	Holt das Zeichen (Item), das sich bei der angegebenen Koordinate im Gameboard befindet.
moveItem()	Bewegt ein Zeichen im Gameboard von einer Koordinate zu einer Koordinate.

5.9.B.1 Zeiger-Variablen und ihre Funktion

Datenmengen liegen im Speicher und haben eine **Adresse**. Ein **Zeiger** ,zeigt‘ auf die Menge, indem er lediglich die Adresse der Menge speichert. Wenn einer Methode eine Datenmenge überliefert werden soll, wird ihr lediglich mit dem Zeiger die Adresse mitgeteilt. Die Datenmenge wird somit nicht kopiert, stattdessen greift die Methode über die Adresse auf die Daten zu. Diese Datenmengen werden durch Variable repräsentiert. Ein Zeiger speichert also nur die

Adresse einer anderen Variablen und leitet somit die Anfrage auf einen Wert weiter. Folgende Abbildung erörtert den Sachverhalt:

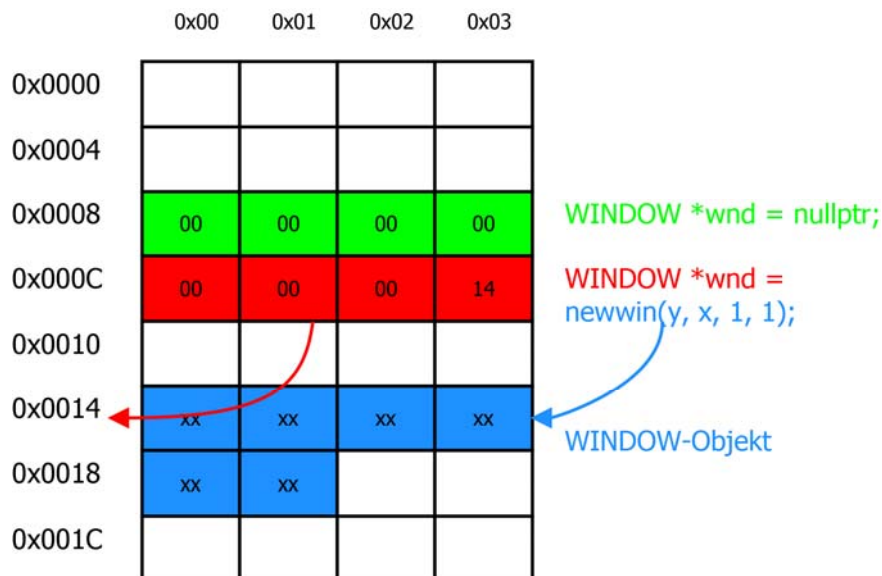


Abbildung 33: Objekte und Zeiger auf diese Objekte im Speicher

Adresse	Erklärung
0x0008	Enthält Zeigervariable <code>wnd</code> initialisiert mit <code>nullptr</code> (0x00000000)
0x000C	Enthält Zeigervariable <code>wnd</code> , die auf ein WINDOW-Objekt zeigt. Der Inhalt von <code>wnd</code> ist die Adresse des Objekts (0x00000014)
0x0014	Das WINDOW-Objekt ist an dieser Adresse gespeichert und ist 6 Bytes lang. Es wird mit der Ncurses-Funktion <code>newwin()</code> angelegt.

5.9.B.2 Die Null-Zeiger-Konstante nullptr

Um eine Zeigervariable z.B. `WINDOW *wnd` auf einen definierten Wert zu setzen, wird das Literal `nullptr` benutzt. In vielen C/C++-Programmen und in der Fachliteratur findet man auch noch zwei andere Möglichkeiten:

0	Die Zahl 0 sollte aus missverständlichen Gründen nur bei Zahlen verwendet werden. Zeigervariablen bitte nicht mit 0 initialisieren.
NULL	Dieses Literal wird sehr häufig verwendet. Es ist, je nach C/C++-Standard und Compiler, unterschiedlich implementiert und somit nicht immer gleich definiert. Zeigervariablen bitte nicht mit 0 initialisieren.

Zeigervariablen immer mit dem Literal `nullptr` initialisieren!

5.9.C Part 07-3 – Restauration der Ziele

Der geübte Entwickler hat festgestellt, dass die Ziele, wenn Kisten über sie geschoben werden, verschwinden. Sie werden also noch nicht restauriert. In 5.7.B.2 sind Möglichkeiten der Restauration beschrieben worden. Hier wird aber eine neue Möglichkeit vorgestellt. Im Prinzip werden die Ziele in einen Vektor geschrieben und nach jeder Bewegung wird der Inhalt des Vektors neu dargestellt. Das UML-Diagramm ist wie folgt um `goals` erweitert:

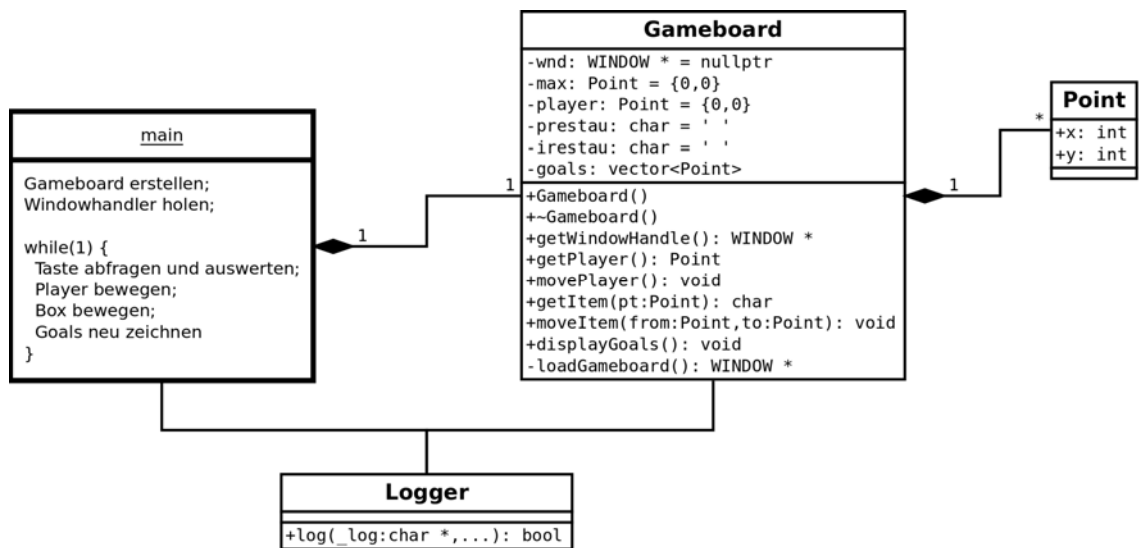


Abbildung 34: erweitertes Klassendiagramm Part 07-3

```

10 #include <vector>
11 #include <memory>
12
110 vector<Point> goals; // Deklaration eines Point-Vektors
116 goals.push_back({1,5}); // Ein Point in den Vektor anhängen
  
```

Zeile	Funktion
10-11	Die benötigten Include-Dateien einbinden
110	Eine Variable <code>goals</code> vom Datentyp <code>vector<Point></code> wird als Eigenschaft deklariert. Hier werden die Ziele gespeichert.
116	Ein Point <code>{1,5}</code> (<code>x=1,y=5</code>) wird in dem Vektor <code>goals</code> gespeichert. Das wird mit der Vektor-Methode <code>push_back()</code> erzielt. Es können beliebig viele Points gespeichert werden.

Sind alle Ziele (3 Stück in Abbildung 31) in den Vektor gespeichert, können mittels einer Range-based for-Schleife (5.3.B.3) alle Ziele ausgegeben werden. Somit kann eine besondere Restauration erzeugt werden.


```

120 for (auto goal : goals) {
121     cout << "x=" << goal.x << ",y=" << goal.y << endl;
122 }

```

Zeile	Funktion
120	<p>Die Range-based for-Schleife:</p> <ul style="list-style-type: none"> • <code>auto goal</code>: in <code>goal</code> ist ein einzelnes Element des Vektors gespeichert, also ein <code>Point</code>. Man könnte auch so schreiben: <code>Point goal</code> • <code>goals</code>: Der eigentliche Vektor <code>goals</code> <p>Die Schleife durchläuft den Vektor vom ersten bis zum letzten Element.</p>
121	Ausgabe des aktuellen <code>Point goal</code> .

Methode	Funktion
<code>loadGameboard()</code>	Zeichnet - wie gehabt - den Level und hier muss der <code>vector</code> mit den Zielen gefüllt werden.
<code>displayGoals()</code>	Zeichnet alle Ziele neu, außer auf ein Ziel steht der Spieler selber oder eine Kiste. Kann der Einfachheit halber nach jeder Spielerbewegung aufgerufen werden.

5.9.C.1 Der Vektor-Container

In C++11 sind diverse abstrakte Datentypen definiert, die Problemlösungen deutlich schneller und eleganter ermöglichen. Einer dieser Datentypen ist der `vector`. Deklariert wird eine `vector`-Variable immer mit `vector` und in den `<>`-Klammern muss ein **primitiver** oder **abstrakter Datentyp** oder eine **beliebige Klasse** definiert werden. Damit ist es möglich, beliebige Datentypen oder Objekte in einem Vektor zu speichern. Daher der Name: **Container**! Der Vektor stellt diverse **Methoden**, die auf ihn angewendet werden können, zur Verfügung. Ein Auszug ist nachfolgend dargestellt:

Methoden	Funktion
<code>begin</code>	Gibt einen <code>Iterator</code> auf das erste Element im <code>vector</code> zurück. Ein <code>Iterator</code> ist im Grunde ein <code>Index</code> wie beim <code>Array</code> .
<code>end</code>	Gibt einen <code>Iterator</code> auf das letzte Element im <code>vector</code> zurück
<code>size</code>	Gibt die Anzahl der Elemente in <code>vector</code> zurück
<code>resize</code>	Verändert die mögliche Anzahl der Elemente in <code>vector</code> . Ist nicht

	zwingend notwendig, aber sorgt für Performance-Gewinn.
<code>empty</code>	Gibt an, ob <code>vector</code> leer ist oder nicht
<code>push_back</code>	Fügt ein Element an das Ende des <code>vectors</code> an
<code>at</code>	Gibt eine Referenz auf das betreffende Element zurück, meist in Verwendung mit einem <code>Iterator</code>
<code>clear</code>	Löscht alle Elemente aus <code>vector</code>

Der große Vorteil eines Vektors gegenüber ein Array ist, dass er die Größe des Speichers für die Elemente selbst verwaltet. Damit sind Speicherzugriffsfehler so gut wie ausgeschlossen. Außerdem ist das Einfügen und Löschen von Elementen besonders einfach.

5.9.D Das Ergebnis

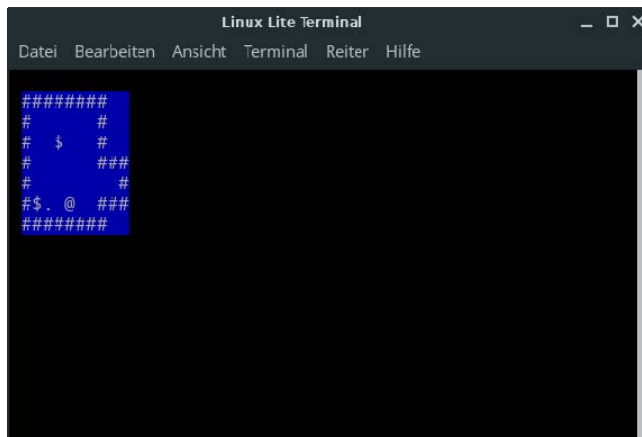


Abbildung 35: Spieler schiebt Kisten auf Ziele inkl. Restauration

5.9.E Triggerpunkt Part 07

An dieser Stelle haben alle Studenten den gleichen Quellcode.

5.10 Part 08 – Aufgabe erfüllt?

Das ist die große Frage. Wenn alle Kisten auf die Ziele verschoben wurden, ist die Aufgabe erfüllt. Das muss dem Spieler mitgeteilt werden. Da kann man sich die ungewöhnlichsten Sachen einfallen lassen. Ihrer Fantasie ist nun keine Grenze gesetzt. **Vorschlag simpel:** Wenn die Aufgabe erfüllt ist, soll das Spiel beendet werden. Es wird eine neue Methode `areGoalsComplete()` benötigt, die prüft, ob alle Kisten auf den Zielen positioniert wurden. In der Endlosschleife in `main()` muss diese Methode aufgerufen werden und – ja nach Rückgabewert – wird das Spiel beendet oder eben nicht:

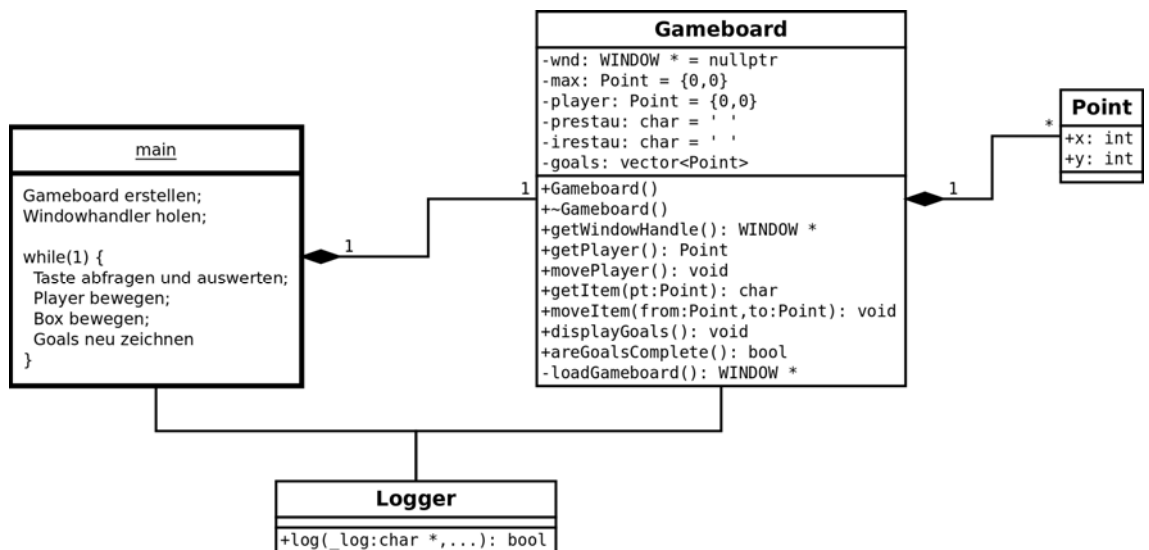


Abbildung 36: erweitertes Klassendiagramm Part 08

Methode	Funktion
<code>areGoalsComplete()</code>	Prüft, ob alle Kisten auf den entsprechenden Zielen stehen. Damit ist die Aufgabe erfüllt. Kann nach jeder Spielerbewegung aufgerufen werden. Besser wäre der Aufruf nach jeder Kistenbewegung.

5.10.A Das Ergebnis

Nachdem alle Kisten auf die Ziele geschoben wurden, muss sich das Spiel beenden.

5.10.B Triggerpunkt Part 08

An dieser Stelle haben alle Studenten den gleichen Quellcode.

Die Spiele-Engine ist mit Part 07 in der einfachsten Form vollständig. Daher erfüllt das Spiel die **Sokoban-Regeln**.

Sind Sie an diesen Punkt angekommen und haben alle Kapitel zuvor eigenständig und mit viel Fleiß und Schweiß erarbeitet (manchmal auch mit Hilfe), können Sie zu Recht stolz auf sich und Ihre Leistung sein.

5.11 Part 09 – Aufgabe laden bitte schön

Die Aufgabe, die in der Vergangenheit genutzt wurde, ist im Quellcode fest einprogrammiert worden. Es ist aber nicht sinnvoll, eine neue Aufgabe immer neu zu programmieren. Stattdessen werden neue Aufgaben **nachgeladen**. Es wird folgender Ablauf, der letztendlich programmiert wird, vorgeschlagen:

1. Erstellen einer neuen Aufgabe mit dem **vi** oder einem anderen **ASCII-Editor** in einer Text-Datei. Zur besseren Verwaltung werden die Dateien mit einer Nummer, einem Text und einem Suffix versehen: `01-name.map`
2. Einlesen der Aufgabe `01-name.map` mittels der modifizierten Methode `loadGameBoard()`
3. Übertragen der Aufgabenkomponenten wie Wände, Kisten, Ziele und den Spieler in das 2-dimensionale Spielfeld auf dem Monitor

5.11.A Erstellen einer neuen Aufgabe – Das Level-Design

Mit dem **vi** wird eine neue beispielhafte Aufgabe wie folgt erstellt

	0	1	2	3	4	5	6	7	8	9
0	#	#	#	#	#	#	#	#		
1	#							#		
2	#		\$		\$			#		
3	#							#	#	#
4	#	#	#	#	\$				@	#
5	#	.	.				.	#	#	#
6	#	#	#	#	#	#	#	#		

Abbildung 37: Beispiel-Aufgabe mit dem **vi** erstellen

und entsprechend unter `01-test.map` abgespeichert. Der Wechsel von einer Zeile zur nächsten wird mit `Enter` bewerkstelligt. Es kann nur einen Spieler

geben. Es können beliebig viele Kisten platziert werden, aber genauso viele Ziele.

5.11.B Part 09-1 – Lesen einer neuen Aufgabe in ein Vektor

Die Aufgabe, die mittels `vi` auf der Festplatte gespeichert wurde, soll in der Methode `loadGameBoard()` geladen werden. Es wird eine Schnittstelle zur Dateibearbeitung benötigt. Die entsprechende Header-Datei lautet:

```
7 #include <fstream> // neue Funktion der File-Streams
```

Nun folgt eine Methode, die als Basis für das Einlesen einer Aufgabe genutzt werden kann. Alle Zeilen der Datei werden in einem Vektor gespeichert:

```
282 WINDOW *Gameboard::loadGameboard() {
283     WINDOW *wnd;
285     vector<string> vec;
286
287     // open stream for reading..
288     fstream f("01-test.map", ios::in);
289     if (f.good()) {
290         while (!f.eof()) {
291             string str;
292             getline(f, str);
293             if (string::npos == str.find(';')) {
294                 vec.push_back(str);
295                 // hier den größten X-Wert bestimmen!
296             }
297         }
298         f.close();
299         // hier den größten Y-Wert bestimmen!
300         ...
320     }
321 }
```

Zeile	Funktion
283-285	Deklaration einiger Variablen, die benötigt werden. In <code>vec</code> werden alle Zeilen der Datei gespeichert
288	Die Aufgabendatei wird geöffnet und in dem Objekt <code>f</code> gespeichert
289	Wenn beim Öffnen der Datei kein Fehler aufgetreten ist: <code>f.good()</code>
290	Solange das Ende der Datei noch nicht erreicht ist: <code>eof()</code> ist eine Methode des Dateistroms <code>f</code> , die das Ende der Datei zurückgibt.
292	<code>getline()</code> holt die aktuelle Zeile der Datei <code>f</code> und speichert den Inhalt in <code>str</code>

293	<code>if</code> -Statement ist erfüllt, wenn kein Semikolon <code>;</code> in der Zeile der Datei <code>f</code> gefunden wurde (<code>;</code> heißt Kommentar in der Aufgabendatei)
294	Speichert die aktuelle Zeile der Datei <code>f</code> in dem vector <code>vec</code>
298	Schließen der Aufgabendatei <code>f</code>
300	Weiter geht's im folgenden Abschnitt

5.11.B.1 Textdateien-Schnittstelle

Im vorherigen Kapitel sind diverse Methoden im Umgang mit Dateien benutzt worden. Die Schnittstelle zu Dateien ist generell besonders wichtig, weil es eine grundlegende Funktion vieler Programme ist. Es sind im folgendem einige Methoden im Umgang mit Dateien aufgeführt, eine vollständige Beschreibung aller Methoden und Operatoren können im C++11-Standard nachgelesen werden.

Methoden	Funktion
<code>fstream f</code>	Öffnet eine Datei <code>filename</code> zum Lesen <code>ios::in</code>
<code>good</code>	Prüft, ob der Status der Datei <code>ok</code> oder fehlerhaft ist
<code>eof</code>	Prüft, ob das Ende einer Datei erreicht ist
<code>getline</code>	Liest ganze Zeile einer Datei ein und speichert den Inhalt in einen String. Die Datei muss eine Textdatei sein, dessen Zeilen mit <code>'\n'</code> enden. Diese Methode wird auch häufig mit <code>cin</code> genutzt. <code>cin</code> ist im Grunde wie eine Textdatei und das abschließende <code>Enter</code> entspricht <code>'\n'</code> .
<code>close</code>	Schließen einer Datei

5.11.B.2 Maximale Anzahl Spalten und Zeilen bestimmen

Alle Aufgaben in Sokoban sind unterschiedlich groß, die Spalten und Zeilen variieren je Aufgabe. Aus diesem Grund ist die Bestimmung der maximalen Spalten und Zeilen wichtig für die Größe des Spielfelds:

Maximale Zeilen Y:	Die Zeilen müssen einfach nur gezählt werden (Stelle siehe Kommentar)
Maximale Spalten X:	Bei den Spalten muss das Maximum gesucht werden, d.h. die Zeile mit der größten Anzahl der Zeichen. Da jede Zeile ein String ist, kann die Anzahl der Zeichen

	mittels der String-Methode <code>length()</code> bestimmt werden (Stelle siehe Kommentar).
--	--

5.11.C Part 09-2 – Zuordnen der Vektor-Elemente

Alle Zeilen der Aufgabendatei sind in dem Vektor `vec` gespeichert und die maximale Anzahl der Zeilen und Spalten (X/Y-Koordinaten) sind bekannt. Jetzt heißt es durch den Vektor zu laufen, die Spielerkoordinate (`player`) und die Ziele (`goals`) zu bestimmen und das Ganze als Spielfeld auf dem Monitor ausgeben.

Zur Erinnerung: in `loadGameBoard()` war vorher auch die Spielerkoordinate `player` und der Ziele-Vektor `goals` festgelegt. **Schauen Sie nach!**

Um durch den Vektor zu laufen und jedes Zeichen auszuwerten, ist eine geschachtelte Range-based for-Schleife sinnvoll:

```

300 for (auto obj: vec) { // obj: Zeile
301     for (auto ch: obj) { // ch: Spalte oder Zeichen
302         switch (ch) {
303             case '@':
304                 ...
305                 break;
306             case '.':
307                 ...
308                 break;
309             default:
310                 break;
311         }
312         mvwaddch(...);
313     }
314 }

```

Zeile	Funktion
300	Laufe durch jede Zeile
301	Laufe durch jede Spalte (Zeichen) einer Zeile
302	Switch-case-Anweisung für jedes Zeichen
303-305	Wenn Spieler gefunden, merke die aktuelle Position in <code>player</code>
306-308	Wenn Ziele gefunden, merke die aktuelle Position in Vektor <code>goals</code>
309-310	Default-Statement immer wichtig, auch wenn es leer ist!
312	Zeichne jedes Zeichen als Spielfeld auf den Monitor. Dafür werden y/x-Koordinaten benötigt, die zuvor in den Range-based for-Schleifen bestimmt werden müssen.

Nun ist das Programm in der Lage, beliebige Aufgaben, die vorher erstellt wurden, zu laden und zu spielen. Ein Wehrmutstropfen ist, dass eine Aufgabe im Quellcode fest programmiert ist und somit nur diese ohne Neukompilieren gespielt werden kann. Das wird im nächsten Abschnitt geändert.

5.11.D Part 09-3 – Datei als Kommandozeilenparameter

Um beliebige Aufgaben spielen zu können, müssen dem Spiel Dateinamen der Aufgaben mitgeteilt werden. Die Aufgaben werden dann dynamisch geladen. Es muss der feste Dateiname in 5.11.B dynamisch mittels einer String-Variablen implementiert werden:

```
288  fstream f("01-test.map", ios::in); // fester Dateiname!
```

Das UML-Diagramm gibt Auskunft, an welchen Stellen der Dateiname (fileName) geändert werden muss:

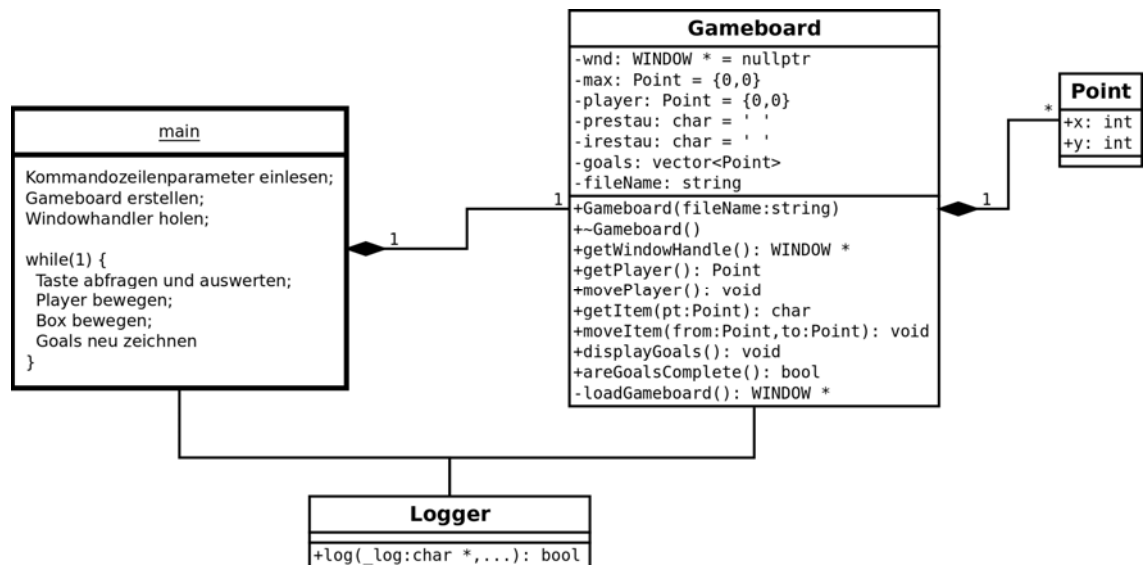


Abbildung 38: erweitertes Klassendiagramm Part 09-3

Der Dateiname muss in der `main()`-Funktion eingelesen werden. Das kann natürlich mit der Einlese-Methode `cin` durchgeführt werden, ist aber zu einfach. Die sog. Kommandozeilenparameter sind da eleganter.

5.11.D.1 Die Kommandozeilenparameter

Kommandozeilenparameter werden dann genutzt, um beim Aufruf des

Programms mit Leerzeichen getrennt Parameter zu übergeben. Es soll folgender Aufruf realisiert werden:

```
./prog 01-test.map
```

01-test.map ist der erste Parameter. Es können aber beliebig viele Parameter mit Leerzeichen getrennt angehängt werden. Um diese Funktionalität zu realisieren, ist folgende Codezeile aus main.cpp gezeigt:

```
109 int main(int argc, char **argv) {
```

main() wird über das Betriebssystem unmittelbar nach dem Aufruf des Spiels aufgerufen. Das Betriebssystem gibt dem Programmaufruf zwei Parameter mit:

int argc	Die Variable argc (<u>A</u> rgument <u>C</u> ount) gibt an, wie viel Argumente übergeben wurden
char **argv	Die Variable argv (<u>A</u> rgument <u>V</u> ector) enthält die Liste der übergebenen Argumente als Strings

Der Speicher, der dafür nötig ist, wird je nach Anzahl der Parameter vom Betriebssystem verwaltet und der Entwickler braucht nichts reservieren:

	0	1	2	3	4	5	6	7	8	9
argv[0]	P	R	O	G	N	A	M	E		
argv[1]	P	A	R	A	M	1				
argv[2]	P	A	R	A	M	2				
argv[3]				

← argv[2][0] ← argv[1][3]

Abbildung 39: Kommandozeilenargumente im Speicher

Das Betriebssystem legt den Aufbau und die Reihenfolge der übergebenen Parameter fest. Die drei großen Betriebssysteme – Linux, Windows und MacOS – definieren das wie in Abbildung 39 dargestellt. Der Parameter argv[0]

enthält immer den Programmnamen, hier `prog`. Das erste Argument ist über `argv[1]` erreichbar. Das zweite über `argv[2]` usw. Der Integerwert `argc` enthält die Anzahl der übergebenen Argumente inkl. Programmname.

Der Aufruf des Programms

```
./prog 01-test.map
```

wird in der Parameterliste von `main` folgendermaßen münden:

- `argc`: 2, Integer
- `argv[0]`: `./prog`, String
- `argv[1]`: `01-test.map`, String
- `argv[2..n]`: nicht gültig und führt u.U. zu einem Programmabbruch

Es fehlt noch die Implementierung. Auf geht's.

5.11.E Das Ergebnis



Abbildung 40: Aufgabe ist übergeben worden und kann gespielt werden

5.11.F Triggerpunkt Part 09

An dieser Stelle haben alle Studenten den gleichen Quellcode.

6. Erweiterung des Computerspiels Sokoban

Dieser Abschnitt ist für die fortgeschrittenen Studenten gedacht und betrifft Erweiterungen des Spiels Sokoban!

6.1 Part 10 – Moves und Pushes

Es ist natürlich eine Herausforderung, eine Aufgabe zu meistern. Zusätzlich gibt es noch Erweiterungen, die Herausforderung zu steigern. Möglichst wenig Züge des Spielers (Moves) oder möglichst wenig Züge des Spielers mit einer Kiste (Pushes) erschweren das Lösen einer Aufgabe.

6.1.A Part 10-1 – Ein zusätzliches Fenster für die Anzeige

Es wird ein neues Fenster benötigt, um Statusanzeigen wie die Moves und Pushes darzustellen. Das neue Fenster soll unterhalb des Spielfelds angeordnet sein. Das neue UML-Klassendiagramm ist wie folgt:

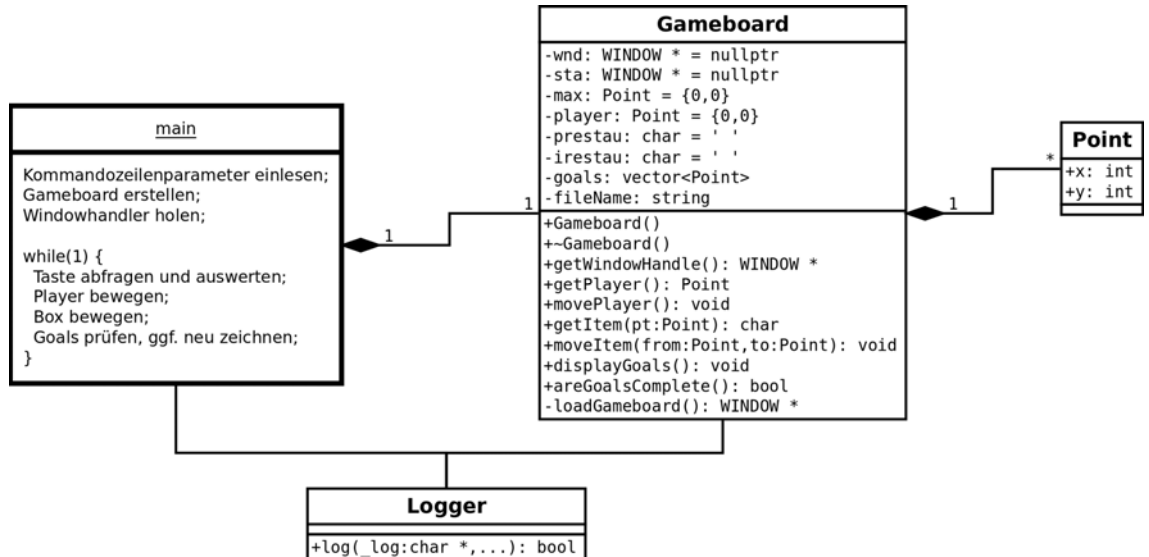


Abbildung 41: erweitertes Klassendiagramm Part 10-1

Das Fenster soll im Moment eine Zeile darstellen können.

Methoden	Funktion
loadGameboard()	Die Methode implementiert das Spielfeld als Fenster. Sie weiß, wie groß das Spielfeld ist. Es liegt nahe, auch hier das Fenster für die Statusanzeige zu erstellen, um es

unterhalb des Spielfensters zu platzieren.

6.1.A.1 Das Ergebnis

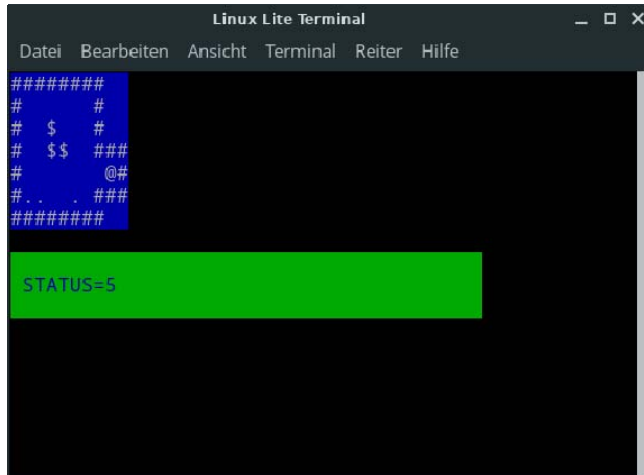


Abbildung 42: Das neue Fenster für die Anzeige von Statusanzeigen

6.1.B Part 10-2 – Anzeige mit Leben füllen

Nun gilt es, das in 6.1.A erzeugte Fenster mit Leben zu füllen, also die Pushes und Moves zu bestimmen und anzuzeigen. Der Quellcode muss folgendermaßen abgeändert werden:

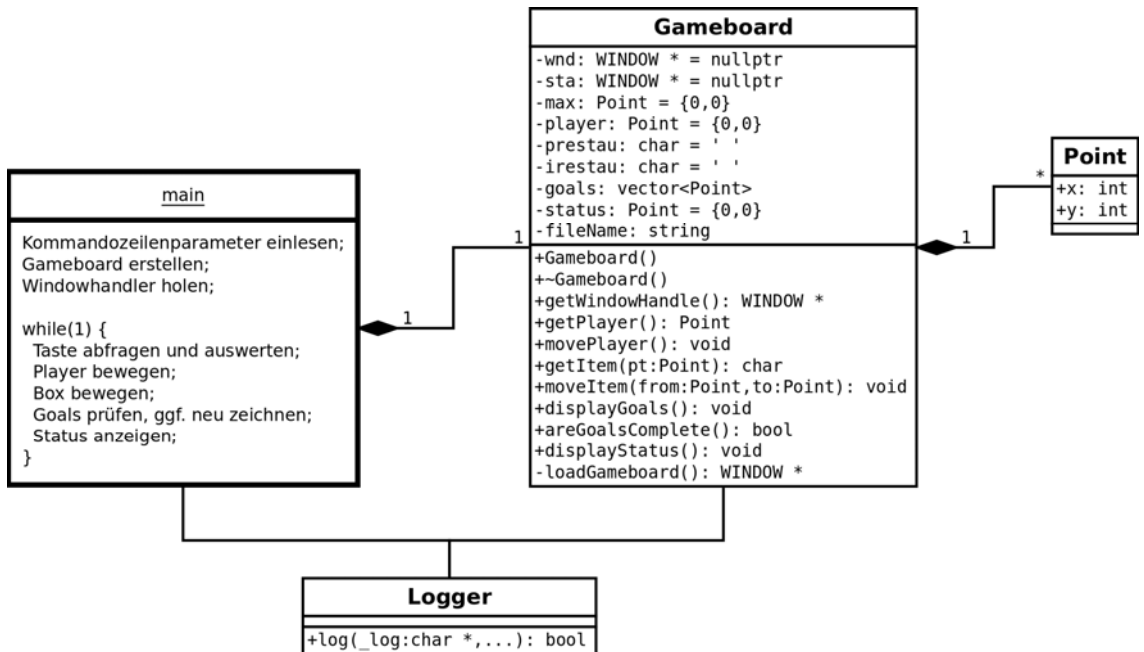


Abbildung 43: erweitertes Klassendiagramm Part 10-2

Die Eigenschaft `status` ist vom Datentyp `Point`. Somit könnte das Element `x`

die Moves und y die Pushes aufnehmen.

Methode	Funktion
displayStatus()	Zeigt den Status in dem Statusfenster an. Diese Methode kann nach jeder Spielerbewegung aufgerufen werden.

6.1.B.1 Das Ergebnis



Abbildung 44: Das Statusfenster mit den Moves und Pushes

6.1.C Triggerpunkt Part 10

An dieser Stelle haben alle Studenten den gleichen Quellcode.

6.2 Part 11 – Eine Ampel als Abwärtszähler

In vielen Spielen ist ein Abwärtszähler Gang und Gäbe. In dem Spiel Sokoban gibt er an, wie lang die Lösung der Aufgabe dauern darf. Eine Ampel, bestehend aus Grün, Gelb und Rot, gibt den Status des Abwärtszähler folgendermaßen an:

- Grün: 70% der Lösungszeit
- Gelb: 20% der Lösungszeit
- Rot: 10% der Lösungszeit
- Aus: Lösungszeit verstrichen

6.2.A Part 11-1 – Die Timer-Threads

Für den Abwärtszähler werden sog. Timer-Threads verwendet. Ein Thread ist ein paralleler Aktivitätsstrang. Mehrere Threads können somit mehrere Aktivitäten parallel ausführen. Ein Timer-Thread ist ein zeitlich gebundener paralleler Aktivitätsstrang. Er wird gestartet und ist eine Zeit lang aktiv. Das UML-Diagramm zeigt die benötigten Eigenschaften und Methoden:

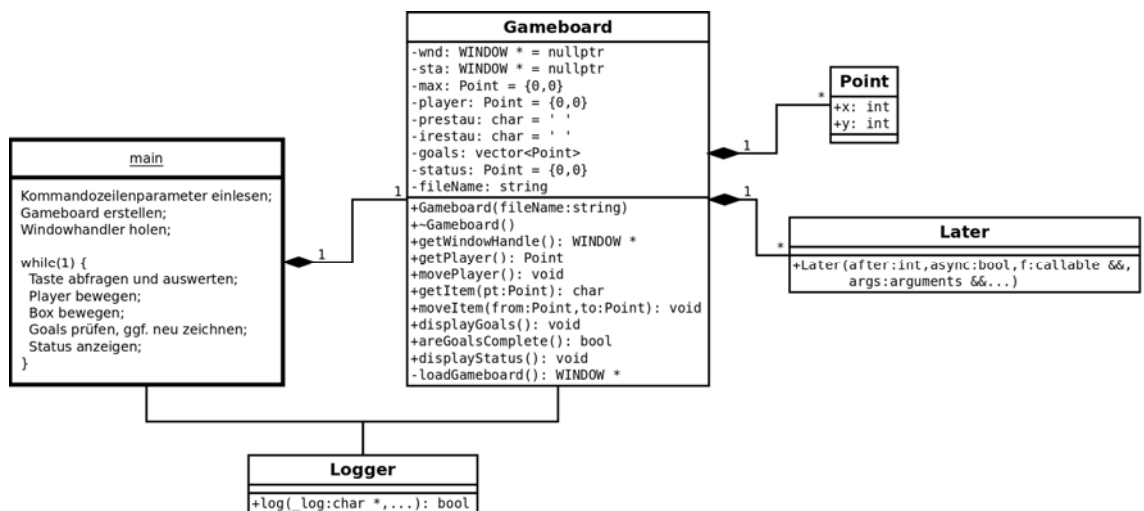


Abbildung 45: erweitertes Klassendiagramm Part 11-1

Das Modul `Later` mit der gleichnamigen Klasse wird zur Verfügung gestellt!

Der folgende Quellcode von `gameboard.cpp` verdeutlicht die Nutzung der

Later-Schnittstelle:

```

20 void testCallback(int arg) {
21     Logger::getInstance()->log((char *)"Callback: %i", arg);
22     return;
23 }
...
28 Gameboard::Gameboard(string fileName) {
29     ...
58     Logger::getInstance()->log((char *)"Aufruf");
59     Later later_test(2000, true, &testCallback, 111);
60 }

```

Zeile	Funktion
28	Die gewohnte Konstruktor-Methode von Gameboard
58	Der Start des Timer-Threads wird in das Logger-Modul geschrieben
59	Die Installation eines Timer-Threads. <code>Later</code> ist die Klasse. <code>later_test</code> ist der Aufruf des Konstruktors der Klasse <code>Later</code> mit 4 Parametern: <ol style="list-style-type: none"> 1. Wartezeit in Millisekunden, bis die Timerfunktion gestartet wird 2. Asynchroner Aufruf, um mehrere Threads parallel laufen zu lassen 3. Timerfunktion, die die Aktivität des Threads implementiert 4. Parameterliste, die der Timerfunktion übergeben wird. → Der Konstruktornamen (<code>later_test</code>) ist frei wählbar!
20	Der Kopf der Timerfunktion des Threads mit Variablen <code>arg</code> , die den übergebenen Parameter übernimmt.
21	Die Aktivität des Threads: Die Variable <code>arg</code> wird in das Logger-Modul geschrieben. Mehr wird hier nicht gemacht.
22-23	Aktivität ist beendet: Timerfunktion ist zu Ende.

Da die Timer-Threads die Thread-Bibliothek benötigen, muss das `Makefile` angepasst werden:

```
3 LDXXFLAGS = -lncurses -lpthread
```

6.2.A.1 Ergebnis

In der Datei `log.txt` sollten folgende Einträge enthalten sein:

```

1 (2019-08-15 12:43:39:890): Aufruf
2 (2019-08-15 12:43:41:896): Callback: 111

```

Die roten Zahlen stellen die Sekunden dar, die zwischen dem Erstellen des

Timer-Threads und dem Aufruf der Timerfunktion vergangen sind, also 2 Sekunden. Genau mit der Wartezeit (**2000 ms**) wurde der Thread erstellt.

6.2.B Part 11-2 – Die Ampel-Timer

Nun wird die Ampel als Abwärtszähler implementiert. Es werden 4 Timer-Threads benötigt:

1. Grün: Start ab 0 Minuten
2. Gelb: Start ab 4 Minuten
3. Grün: Start ab 5 Minuten
4. Aus: Start ab 6 Minuten

Bevor die Timer-Threads im Gameboard-Konstruktor gestartet werden, müssen die drei Farben initialisiert werden:

```

57 Gameboard::Gameboard(string fileName) {
58     ...
87     init_pair(3, COLOR_WHITE, COLOR_GREEN);
88     ...
89     // Timer-Threads initialisieren
90     ...
95 }

```

Danach können die Timer-Threads nach UML-Klassendiagramm implementiert werden:

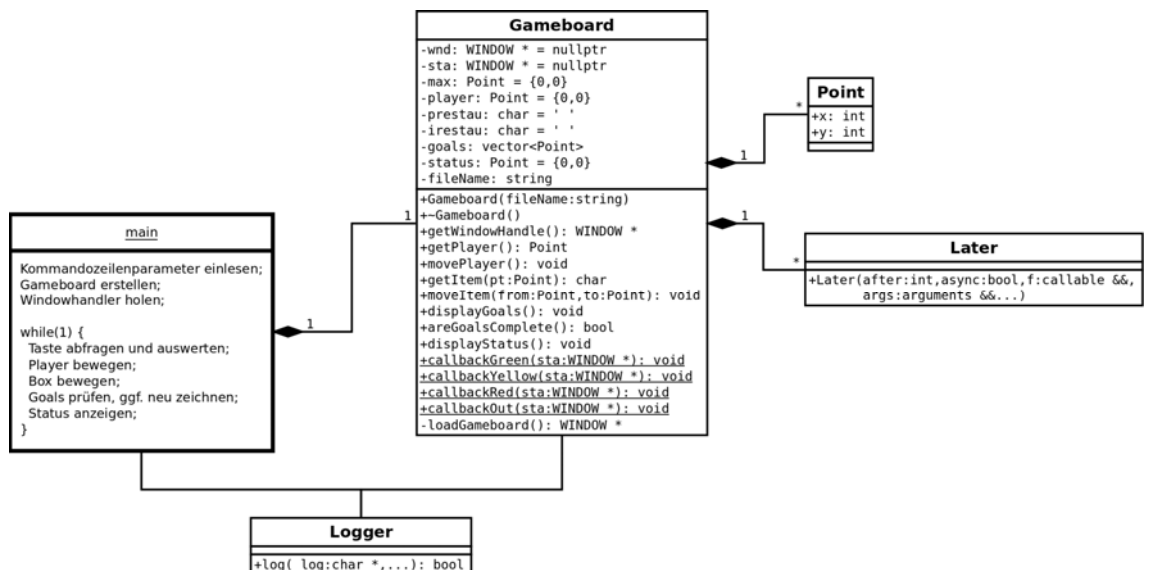


Abbildung 46: erweitertes Klassendiagramm Part 11-2

Die im UML-Klassendiagramm unterstrichenden Methoden sind **statisch**, d.h. eine statische Methode ist ein Merkmal der Klasse, nicht der Objekte, die sie erzeugt hat. Sie existieren für alle Objekt der Klasse nur einmal.
 → Da es nur ein Gameboard-Objekt gibt, sind die statischen Methoden auch Merkmal dieses einen Objekts!

Da die statischen Timerfunktionen auf das Status-Fenster zugreifen müssen, muss das Fensterobjekt WINDOW *sta als Parameter allen Timerfunktionen mitgegeben werden. In den Timerfunktionen müssen dann die jeweiligen Farben erstellt und gelöscht werden (hier Ausschnitt der Timerfunktion Gelb):

```
27 mvwaddch(sta, 1, 27, ' ' | COLOR_PAIR(4) | A_BOLD);
```

Zeile	Funktion
27	Setze ein Leerzeichen mit der Farbe COLOR_PAIR(4) und zusätzlich fett auf Position 1,27. COLOR_PAIR(4) ist Gelb

Somit ist das weitere Vorgehen bezüglich der Timer-Funktionen ableitbar.

Methode	Funktion
callbackGreen()	Zeigt im Statusfenster die Grünphase an
callbackYellow()	Zeigt im Statusfenster die Gelbphase an
callbackRed()	Zeigt im Statusfenster die Rotphase an

6.2.B.1 Ergebnis

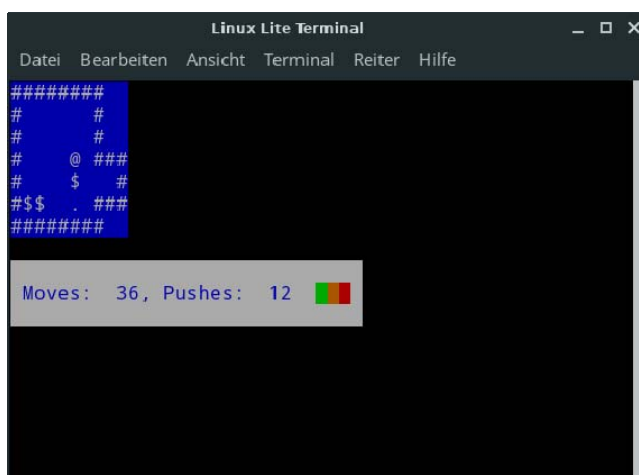


Abbildung 47: Die Status-Ampel in Aktion

6.2.C Part 11-3 – Wartezeit per Kommandozeile

Die Wartezeit für die Ampelfarben ist bisher fest programmiert. Nun soll die Gesamtzeit dem Programm per Kommandozeilenparameter übergeben werden:

- Grün: Start ab 0 Minuten
- Gelb: Start ab 70% der Wartezeit
- Rot: Start ab 90% der Wartezeit
- Aus: Start bei 100% der Wartezeit

Das Hauptprogramm muss um den Kommandozeilenparameter Wartezeit erweitert werden. Es sind damit 2 Kommandozeilenparameter. Die Wartezeiten der einzelnen Timer-Threads müssen obigen Vorgaben entsprechend angepasst werden:

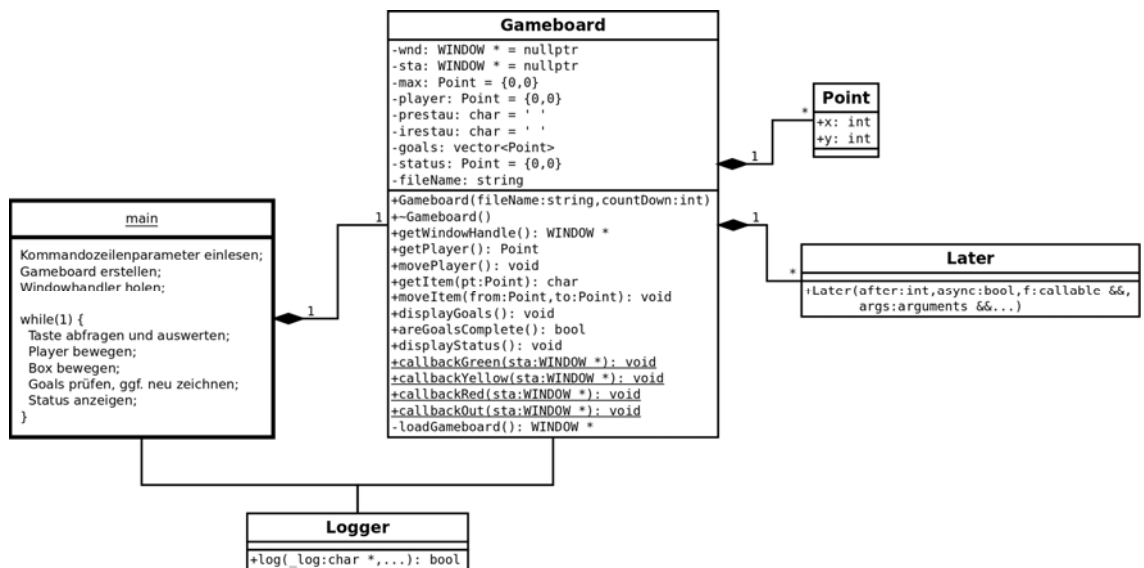


Abbildung 48: erweitertes Klassendiagramm Part 11-3

6.2.D Triggerpunkt Part 11

An dieser Stelle haben alle Studenten den gleichen Quellcode.

6.3 Part 12 – Spieler, Kisten und Ziele einfärben

Der Spieler, die Kisten und die Ziele sind bisher in Weiß gehalten. Sie sollen in diesem Abschnitt eingefärbt werden:

- Spieler: soll rot eingefärbt werden
- Kisten: sollen blaugrün (cyan) eingefärbt werden
- Ziele: sollen gelb eingefärbt werden

An dieser Stelle sind alle theoretischen Informationen vorhanden, um diesen Abschnitt ohne nennenswerte Hilfe umzusetzen.

6.3.A.1 Ergebnis

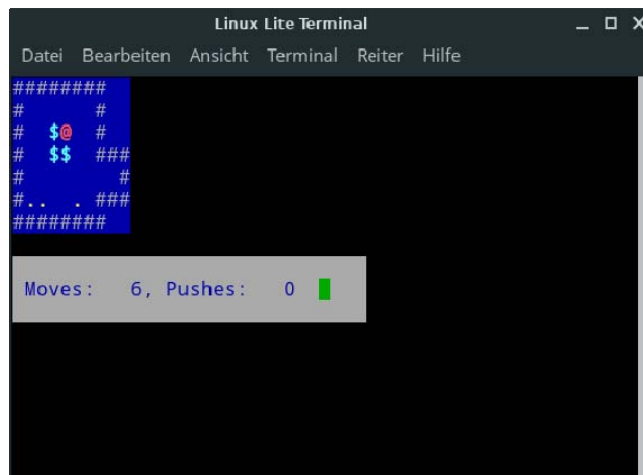


Abbildung 49: Spieler, Kisten und Ziele eingefärbt

6.3.A.2 Triggerpunkt Part 12

An dieser Stelle haben alle Studenten den gleichen Quellcode.

7. Zusatz – Erweiterungen

Dieser Abschnitt beschäftigt sich mit Ideen, die als zusätzliche Erweiterungen verstanden werden können. Die nachfolgende Tabelle listet einige Ideen auf.

#	Idee	Beschreibung
1	Aufnahme- und Wiedergabefunktion	<p><u>Sinn:</u> Lösungsschritte können veröffentlicht werden.</p> <p><u>Durchführung:</u> Durch einen Tastendruck (τ) wird eine Aufnahme der Züge in eine Datei gestartet, die bei gleichem Tastendruck beendet wird. Die Tasten w, a, s, und d werden nacheinander in die Datei geschrieben. Mit einem Tastendruck (p) wird die Aufnahme wiedergegeben.</p>
2	Teleporter	<p><u>Sinn:</u> Aufgaben können komplexer erstellt werden</p> <p><u>Durchführung:</u> Durch Teleporter, die mit Nummern gekennzeichnet werden können, ist der Spieler in der Lage, unterschiedliche Aufgabenabschnitte zu erreichen</p>
2	Undo-Funktion	<p><u>Sinn:</u> Ein unbeabsichtigter Zug kann rückgängig gemacht werden.</p> <p><u>Durchführung:</u> Durch einen Tastendruck (u) wird der letzte Zug mit all seinen Auswirkungen auf dem Spielfeld rückgängig gemacht. Diese Funktion kann noch auf alle Züge erweitert werden.</p>
3	Separierte Ziele	<p><u>Sinn:</u> Erhöhung des Schwierigkeitsgrad des Spiels</p> <p><u>Durchführung:</u> Ziele und Kisten werden farbig dargestellt. Die</p>

		Kisten müssen dann auf die Ziele geschoben werden, die farblich zu ihnen passen. Das Aufgabenformat muss entsprechend geändert werden.
4	Sokoban über Internet-Browser	<u>Sinn:</u> Spiel online spielen <u>Durchführung:</u> Spiel über Wetty oder goTTY über Web-Browser onlinefähig machen.
5	Spielstände zentral speichern	<u>Sinn:</u> Spielstände manipulationssicher und zentral speichern. <u>Durchführung:</u> Die Spielstände mit Moves, Pushes und der Spielzeit speichern. Ein zentrales DBMS muss dafür eingerichtet werden. Hier bieten sich MariaDB oder PostgreSQL an. Eine Schnittstelle vom Spiel (C++) zum DBMS muss implementiert werden.

8. Anlagen

8.1 ASCII-Tabelle

000	NUL	033	!	066	B	099	c	132	ä	165	Ñ	198	ã	231	þ
001	Start Of Header	034	"	067	C	100	d	133	á	166	·	199	Ä	232	Ë
002	Start Of Text	035	#	068	D	101	e	134	â	167	°	200	Å	233	Ü
003	End Of Text	036	\$	069	E	102	f	135	ç	168	¿	201	Æ	234	Ù
004	End Of Transmission	037	%	070	F	103	g	136	ê	169	®	202	⚡	235	Ú
005	Enquiry	038	&	071	G	104	h	137	ë	170	¬	203	⚡	236	Ý
006	Acknowledge	039		072	H	105	i	138	è	171	½	204	⚡	237	Ÿ
007	Bell	040	(073	I	106	j	139	í	172	¼	205	=	238	˘
008	Backspace	041)	074	J	107	k	140	î	173	ı	206	⚡	239	˙
009	Horizontal Tab	042	*	075	K	108	l	141	ï	174	«	207	⚡	240	-
010	Line Feed	043	+	076	L	109	m	142	Ā	175	»	208	δ	241	±
011	Vertical Tab	044	,	077	M	110	n	143	Ā	176	⋮	209	⊕	242	₊
012	Form Feed	045	-	078	N	111	o	144	É	177	⋮	210	Ê	243	¼
013	Carriage Return	046	.	079	O	112	p	145	æ	178	⚡	211	Ë	244	⚡
014	Shift Out	047	/	080	P	113	q	146	Æ	179		212	Ë	245	§
015	Shift In	048	0	081	Q	114	r	147	ô	180	↓	213	ı	246	÷
016	Delete	049	1	082	R	115	s	148	ö	181	Ā	214	ı	247	,
017	-- frei --	050	2	083	S	116	t	149	ò	182	Ā	215	ı	248	°
018	-- frei --	051	3	084	T	117	u	150	ú	183	Ā	216	ı	249	˘
019	-- frei --	052	4	085	U	118	v	151	ù	184	⊗	217	⚡	250	.
020	-- frei --	053	5	086	V	119	w	152	ÿ	185	⚡	218	⚡	251	˘
021	Negative Acknowledge	054	6	087	W	120	x	153	Ö	186		219	■	252	˘
022	Synchronous Idle	055	7	088	X	121	y	154	Û	187	⚡	220	■	253	˘
023	End Of Transmission Block	056	8	089	Y	122	z	155	ø	188	⚡	221	ı	254	■
024	Cancel	057	9	090	Z	123	{	156	£	189	⚡	222	ı	255	■
025	End Of Medium	058	:	091	[124		157	∅	190	¥	223	■		
026	Substitute	059	;	092	\	125	}	158	×	191	⚡	224	Ó		
027	Escape	060	<	093]	126	~	159	f	192	⚡	225	ß		
028	File Separator	061	=	094	^	127	▯	160	á	193	⚡	226	Ô		
029	Group Separator	062	>	095	_	128	Ç	161	í	194	⚡	227	Õ		
030	Record Separator	063	?	096	`	129	ü	162	ó	195	⚡	228	ö		
031	Unit Separator	064	@	097	a	130	é	163	ú	196	—	229	Û		
032		065	A	098	b	131	â	164	ñ	197	⚡	230	μ		

Abbildung 50: ASCII-Code Tabelle

8.2 C++ Referenzkarte

C++ Reference Card

C++ Data Types

Data Type	Description
bool	boolean (true or false)
char	character ('a', 'b', etc.)
char[]	character array (C-style string if null terminated)
string	C++ string (from the STL)
int	integer (1, 2, -1, 1000, etc.)
long int	long integer
float	single precision floating point
double	double precision floating point

These are the most commonly used types; this is not a complete list.

Operators

The most commonly used operators in order of precedence:

1	++ (post-increment), -- (post-decrement)
2	! (not), ++ (pre-increment), -- (pre-decrement)
3	*, /, % (modulus)
4	+, -
5	<, <=, >, >=
6	== (equal-to), != (not-equal-to)
7	&& (and)
8	(or)
9	= (assignment), *=, /=, %=, +=, -=

Console Input/Output

```
cout <<          console out, printing to screen
cin >>          console in, reading from keyboard
cerr <<        console error
```

```
Example:
cout << "Enter an integer: ";
cin >> i;
cout << "Input: " << i << endl;
```

File Input/Output

```
Example (input):
ifstream inFile;
inFile.open("data.txt");
inFile >> inputVariable;
// you can also use get (char) or
// getline (entire line) in addition to >>
...
inFile.close();
```

```
Example (output):
ofstream outFile;
outFile.open("output.txt");
outFile << outputVariable;
...
outFile.close();
```

Decision Statements

if if (expression) statement;	Example if (x < y) cout << x;
if/else if (expression) statement; else statement;	Example if (x < y) cout << x; else cout << y;
switch/case switch(int expression) { case int-constant: statement(s); break; case int-constant: statement(s); break; default: statement; }	Example switch(choice) { case 0: cout << "Zero"; break; case 1: cout << "One"; break; default: cout << "What?"; }

Looping

```
while Loop
while (expression)
statement;
```

```
Example
while (x < 100)
cout << x++ << endl;
```

```
while (expression)
{
statement;
statement;
}
```

```
while (x < 100)
{
cout << x << endl;
x++;
}
```

do-while Loop

```
do
statement;
while (expression);
```

```
Example
do
cout << x++ << endl;
while (x < 100);
```

```
do
{
statement;
statement;
}
while (expression);
```

```
do
{
cout << x << endl;
x++;
}
while (x < 100);
```

for Loop

```
for (initialization; test; update)
statement;
```

```
for (initialization; test; update)
{
statement;
statement;
}
```

```
Example
for (count = 0; count < 10; count++)
{
cout << "count equals: ";
cout << count << endl;
}
```

Functions

Functions return at most one value. A function that does not return a value has a return type of void. Values received by a function are called parameters.

```
return_type function(type p1, type p2, ...)
{
statement;
statement;
...
}
```

```
Example
int timesTwo(int v)
{
int d;
d = v * 2;
return d;
}
```

```
void printCourseNumber()
{
cout << "CSEL204" << endl;
return;
}
```

Passing Parameters by Value

return_type function(type p1)
Variable is passed into the function but changes to p1 are not passed back.

return_type function(type p1)
Variable is passed into the function and changes to p1 are passed back.

Default Parameter Values

return_type function(type p1=val)
val is used as the value of p1 if the function is called without a parameter.

Pointers

A pointer variable (or just pointer) is a variable that stores a memory address. Pointers allow the indirect manipulation of data stored in memory.

Pointers are declared using *. To set a pointer's value to the address of another variable, use the & operator.

```
Example
char c = 'a';
char* cPtr;
cPtr = &c;
```

Use the indirection operator (*) to access or change the value that the pointer references.

```
Example
// continued from example above
*cPtr = 'b';
cout << *cPtr << endl; // prints the char b
cout << c << endl; // prints the char b
```

Array names can be used as constant pointers, and pointers can be used as array names.

```
Example
int numbers[]={20, 20, 30, 40, 50};
int* numPtr = numbers;
cout << numbers[0] << endl; // prints 20
cout << *numPtr << endl; // prints 20
cout << numbers[1] << endl; // prints 20
cout << *(numPtr + 1) << endl; // prints 20
cout << numPtr[2] << endl; // prints 30
```

Dynamic Memory

Allocate Memory
ptr = new type;
Example
int* iPtr;
iPtr = new int;

ptr = new type[size];
int* intArray;
intArray = new int[5];

Deallocate Memory
delete ptr;
delete [] ptr;
Example
delete iPtr;
delete [] intArray;

Once a pointer is used to allocate the memory for an array, array notation can be used to access the array locations.

```
Example
int* intArray;
intArray = new int[5];
intArray[0] = 23;
intArray[1] = 32;
```

Structures

Declaration
struct name
{
type1 element1;
type2 element2;
};
Example
struct Hamburger
{
int patties;
bool cheese;
};

Definition
name varName;
Example
Hamburger* hPtr;
hPtr = h;

Accessing Members
varName.element-not;
Example
h.patties = 2;
h.cheese = true;

ptrName->element-not;
hPtr->patties = 1;
hPtr->cheese = false;

Structures can be used (just like the built-in data types) in arrays.

Classes

Declaration

```
class classname
{
public:
    classname(parameters);
    ~classname();
    type member1;
    type member2;
protected:
    type member3;
private:
    type member4;
};
```

public members are accessible from anywhere the class is visible.

private members are only accessible from the same class or a friend (function or class).

protected members are accessible from the same class, derived classes, or a friend (function or class).

constructors may be overloaded just like any other function. You can define two or more constructors as long as each constructor has a different parameter list.

Definition of Member Functions

```
return_type classname::functionname(parameters)
{
    statements;
}
```

Examples

```
Square::Square()
{
    width = 0;
}

void Square::setWidth(float w)
{
    if (w >= 0)
        width = w;
    else
        exit(-1);
}

float Square::getArea()
{
    return width*width;
}
```

Definition of Instances

```
classname varName;

classname* ptrName;
Sptr=new Square(1.0);
```

Accessing Members

```
varName.member-var;
varName.member();

ptrName->member-var;
ptrName->member();
```

Inheritance

Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.

Example

```
class Student
{
public:
    Student(string n, string id);
    void print();
protected:
    string name;
    string netID;
};

class GradStudent : public Student
{
public:
    GradStudent(string n, string id,
                string prev);
    void print();
protected:
    string prevDegree;
};
```

Visibility of Members after Inheritance

Inheritance Specification	Access Specifier in Base Class		
	private	protected	public
private	-	private	private
protected	-	protected	protected
public	-	protected	public

Exceptions

Example

```
try
{
    // code here calls functions that might
    // throw exceptions
    quotient = divide(num1, num2);

    // or this code might test and throw
    // exceptions directly
    if (num3 < 0)
        throw -1; // exception to be thrown can
                // be a value or an object
}
catch (int)
{
    cout << "num3 can not be negative!";
    exit(-1);
}
catch (char* exceptionString)
{
    cout << exceptionString;
    exit(-2);
}
// add more catch blocks as needed
```

Function Templates

Example

```
template <class T>
T getMax(T a, T b)
{
    if (a>b)
        return a;
    else
        return b;
}
```

// example calls to the function template
int a=9, b=1, c;
c = getMax(a, b);

float f=5.3, g=9.7, h;
h = getMax(f, g);

Operator Overloading

C++ allows you to define how standard operators (+, -, *, etc.) work with classes that you write. For example, to use the operator + with your class, you would write a function named operator+ for your class.

Example

Prototype for a function that overloads + for the Square class:
Square operator+ (const Square &);

If the object that receives the function call is not an instance of a class that you wrote, write the function as a friend of your class. This is standard practice for overloading << and >>.

Example

Prototype for a function that overloads << for the Square class:
friend ostream & operator<< (ostream &, const Square &);

Make sure the return type of the overloaded function matches what C++ programmers expect. The return type of relational operators (<, >, ==, etc.) should be bool, the return type of << should be ostream &, etc.

Class Templates

Example

```
template <class T>
class Point
{
public:
    Point(T x, T y);
    void print();
    double distance(Point<T> p);
private:
    T xs;
    T ys;
};
```

// examples using the class template
Point<int> p1(3, 2);
Point<float> p2(3.5, 2.5);
p1.print();
p2.print();

Suggested Websites

C++ Reference: <http://www.cppreference.com/> <http://www.informit.com/golibs/golibs.aspx?g=cplusplus>

C++ Tutorial: <http://www.cplusplus.com/doc/tutorial/> <http://www.sparrowkites.com/cs/>

C++ Examples: <http://www.frostsmarus.com/notes-cpp/>

Gaddis Textbook
Video Notes https://media.pearsoncmg.com/api/v1/media_source_9780130354196/videobooks
Source Code [ftp://ftp.au.com/cseng/authors/gaddis/CCSOS \(5th edition\)](ftp://ftp.au.com/cseng/authors/gaddis/CCSOS%205th%20edition)

Developed for Mississippi State University's CSE1204 and CSE1304 courses
Download the latest version from <http://cse.missstate.edu/~cscrupton/reference>

February 17, 2009

Abbildung 51: C++ Referenzkarte

8.3 GDB Referenzkarte

GDB cheatsheet - page 1

Running

```
# gdb <program> [core dump]
Start GDB (with optional core dump).

# gdb --args <program> <args...>
Start GDB and pass arguments

# gdb --pid <pid>
Start GDB and attach to process.

set args <args...>
Set arguments to pass to program to
be debugged.

run
Run the program to be debugged.

kill
Kill the running program.
```

Breakpoints

```
break <where>
Set a new breakpoint.

delete <breakpoint#>
Remove a breakpoint.

clear
Delete all breakpoints.

enable <breakpoint#>
Enable a disabled breakpoint.

disable <breakpoint#>
Disable a breakpoint.
```

Watchpoints

```
watch <where>
Set a new watchpoint.

delete/enable/disable <watchpoint#>
Like breakpoints.
```

<where>

```
function_name
Break/watch the named function.

line_number
Break/watch the line number in the cur-
rent source file.

file:line_number
Break/watch the line number in the
named source file.
```

Conditions

```
break/watch <where> if <condition>
Break/watch at the given location if the
condition is met.
Conditions may be almost any C ex-
pression that evaluate to true or false.

condition <breakpoint#> <condition>
Set/change the condition of an existing
break- or watchpoint.
```

Examining the stack

```
backtrace
where
Show call stack.

backtrace full
where full
Show call stack, also print the local va-
riables in each frame.

frame <frame#>
Select the stack frame to operate on.
```

Stepping

```
step
Go to next instruction (source line), di-
ving into function.
```

next

Go to next instruction (source line) but
don't dive into functions.

finish

Continue until the current function re-
turns.

continue

Continue normal execution.

Variables and memory

```
print/format <what>
Print content of variable/memory locati-
on/register.

display/format <what>
Like „print“, but print the information
after each stepping instruction.

undisplay <display#>
Remove the „display“ with the given
number.

enable display <display#>
disable display <display#>
En- or disable the „display“ with the gi-
ven number.

x/nfu <address>
Print memory.
n: How many units to print (default 1).
f: Format character (like „print“).
u: Unit.

Unit is one of:
b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).
```

GDB cheatsheet - page 2	
<p>Format</p> <p>a Pointer.</p> <p>c Read as integer, print as character.</p> <p>d Integer, signed decimal.</p> <p>f Floating point number.</p> <p>o Integer, print as octal.</p> <p>s Try to treat as C string.</p> <p>t Integer, print as binary ($t = \text{"two"}$).</p> <p>u Integer, unsigned decimal.</p> <p>x Integer, print as hexadecimal.</p> <p><what></p> <p>expression</p> <p>Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).</p> <p>file_name::variable_name</p> <p>Content of the variable defined in the named file (static variables).</p> <p>function::variable_name</p> <p>Content of the variable defined in the named function (if on the stack).</p> <p>{type}address</p> <p>Content at <i>address</i>, interpreted as being of the C type <i>type</i>.</p> <p>\$register</p> <p>Content of named register. Interesting registers are %esp (stack pointer), %ebp (frame pointer) and %eip (instruction pointer).</p> <p>Threads</p> <p>thread <thread#></p> <p>Chose thread to operate on.</p>	<p>Manipulating the program</p> <p>set var <variable_name>=<value></p> <p>Change the content of a variable to the given value.</p> <p>return <expression></p> <p>Force the current function to return immediately, passing the given value.</p> <p>Sources</p> <p>directory <directory></p> <p>Add <i>directory</i> to the list of directories that is searched for sources.</p> <p>list</p> <p>list <filename>:<function></p> <p>list <filename>:<line_number></p> <p>list <first>,<last></p> <p>Shows the current or given source context. The filename may be omitted. If <i>last</i> is omitted the context starting at <i>start</i> is printed instead of centered around it.</p> <p>set listsize <count></p> <p>Set how many lines to show in „list“.</p> <p>Signals</p> <p>handle <signal> <options></p> <p>Set how to handle signals. Options are: <i>(no)print</i>. (Don't) print a message when signals occurs.</p> <p><i>(no)stop</i>: (Don't) stop the program when signals occurs.</p> <p><i>(no)pass</i>: (Don't) pass the signal to the program.</p>
<p>Informations</p> <p>disassemble</p> <p>Disassemble the current function or given location.</p> <p>info args</p> <p>Print the arguments to the function of the current stack frame.</p> <p>info breakpoints</p> <p>Print informations about the break- and watchpoints.</p> <p>info display</p> <p>Print informations about the „displays“.</p> <p>info locals</p> <p>Print the local variables in the currently selected stack frame.</p> <p>info sharedlibrary</p> <p>List loaded shared libraries.</p> <p>info signals</p> <p>List all signals and how they are currently handled.</p> <p>info threads</p> <p>List all threads.</p> <p>show directories</p> <p>Print all directories in which GDB searches for source files.</p> <p>show listsize</p> <p>Print how many are shown in the „list“ command.</p> <p>what is variable_name</p> <p>Print type of named variable.</p>	<p>Manipulating the program</p> <p>set var <variable_name>=<value></p> <p>Change the content of a variable to the given value.</p> <p>return <expression></p> <p>Force the current function to return immediately, passing the given value.</p> <p>Sources</p> <p>directory <directory></p> <p>Add <i>directory</i> to the list of directories that is searched for sources.</p> <p>list</p> <p>list <filename>:<function></p> <p>list <filename>:<line_number></p> <p>list <first>,<last></p> <p>Shows the current or given source context. The filename may be omitted. If <i>last</i> is omitted the context starting at <i>start</i> is printed instead of centered around it.</p> <p>set listsize <count></p> <p>Set how many lines to show in „list“.</p> <p>Signals</p> <p>handle <signal> <options></p> <p>Set how to handle signals. Options are: <i>(no)print</i>. (Don't) print a message when signals occurs.</p> <p><i>(no)stop</i>: (Don't) stop the program when signals occurs.</p> <p><i>(no)pass</i>: (Don't) pass the signal to the program.</p>

Abbildung 52: GDB Referenzkarte