

Skript zum Vorkurs “Programmierung in C++”

Wintersemester 2024/2025

Ulrich Brenner

Dieses Skript verwendet Material aus früheren Skripten zu Programmierkursen, die von der Fachschaft Mathematik gehalten wurden. Autoren dieser früheren Versionen sind:

Jesko Hüttenhain,
Lars Wallenborn,
Clelia Albrecht,
Felix Boes und
Johannes Holke.

Die Abschnitte 3.8 und 3.9. stammen von Valentin von Bornhaupt.

Vorläufige Version

Inhaltsverzeichnis

1	Informationen zum Vorkurs	4
2	Einführung	5
2.1	Hello World	5
2.2	Der Speicher	5
2.3	Maschinencode und Kompilierung	6
2.4	Die Standardbibliothek	7
2.5	C und C++	8
3	Programmierungsumgebungen	9
3.1	Nutzung der PC-Pools	9
3.2	g++ unter Windows	9
3.3	Architektur (32bit oder 64bit?)	9
3.4	Cygwin installieren	9
3.5	Eine fertige Cygwin-Version	12
3.6	PATH setzen	13
3.7	Bedienung von Cygwin	14
3.8	Installation unter Mac OS X	15
3.9	Bedienen von g++ unter Mac OS X	16
4	Elementare Sprachkonstrukte	20
4.1	Kommentare	20
4.2	Variablen	20
4.3	Operatoren und Ausdrücke	23
4.4	Dateneingabe über die Konsole	25
4.5	Konstanten	25
4.6	If-Else-Statement	26
4.7	Logische und Vergleichsoperatoren	26
4.8	Der Schleifen erster Teil: while	28
4.9	Der Schleifen zweiter Teil: for	28
4.10	Eigene Bezeichnungen für Datentypen	30
4.11	Laufzeitmessung	31
4.12	Zufallszahlen	32
5	Funktionen	33
5.1	Funktionsdefinitionen	33
5.2	Variablenübergabe per Referenz	35
5.3	Funktionsdeklaration vs. Funktionsdefinition	37
5.4	Modulares Programmieren und Linken	38
5.5	Der Präprozessor	41
5.5.1	Makrodefinition	41
5.5.2	Bedingte Texte	41
5.6	Namensräume	42

6	Speicherverwaltung	44
6.1	Aufbau des Speichers	44
6.2	Dynamische Speicherverwaltung mit der Klasse vector	44
6.2.1	Grundlegendes	44
6.2.2	Schleifen über Vektoren	47
6.2.3	Vektoren als Argumente von Funktionen	47
6.2.4	Vektoren als Rückgabe-Werte von Funktionen	49
7	Klassen	50
7.1	Grundbegriffe	50
7.2	Modularisierung mit Klassen	56
7.3	Weiteres zu Konstruktoren	57
7.4	Destruktoren	59
7.5	static-Variablen	59
7.6	Objekte als Klassenelemente	61
7.7	Operatoren	62
7.8	Templates	64

1 Informationen zum Vorkurs

Zeitraum: 23.9.2024 bis 4.10.2024. Jeweils von 10 bis 12 Uhr gibt es Vorlesungen im Großen Hörsaal der Mathematik, Wegelerstraße 10. Nachmittags gibt es von 13 bis 17 Uhr Übungen, die im (Kleinen oder Großen) Hörsaal und in den PC-Pools in der Wegelerstraße 6 (HRZ) und im Nebengebäude des Mathematikzentrums (Endenicher Allee 60) stattfinden.

Dieses Skript basiert auf Skripten, die von der Fachschaft Mathematik in den vergangenen Jahren für C-Vorkurse erstellt wurden. Insbesondere stammen die Bilder aus diesen Skripten.

Das Skript wird während des Programmierkurses laufend aktualisiert, die jeweils neueste Version findet sich auf der PreCampus-Homepage der Veranstaltung:

https://precampus.uni-bonn.de/goto.php?target=crs_78&client_id=precampus

Dort werden im Laufe des Kurses auch weitere Materialien wie Programmbeispiele und die Übungsaufgaben hochgeladen.

Am Ende des Skriptes finden Sie einige Bücher zum Thema C++. Die Bücher von Breymann [2012], Koenig und Moo [2003], Louis [2018], Scheinerman [2006], Theis [2020] und Wolf und Guddat [2022] richten sich an Einsteiger und sind überwiegend recht leicht verständlich, allerdings nicht in allen Teilen präzise. Die drei Bücher von Stroustrup (Stroustrup [2013], Stroustrup [2014], Stroustrup [2023]) zeichnen sich durch große Genauigkeit aus, sind aber vielleicht eher für fortgeschrittene Programmierer zu empfehlen. Im Buch von Hougardy und Vygen [2018] wird die C++-Programmierung, wie sie in der Vorlesung “Algorithmische Mathematik I” erwartet wird, präsentiert.

Es gibt außerdem eine Vielzahl von Internetseiten, auf denen man sich über C++ informieren kann. Eine leicht zu lesende nach dem “Wiki-Prinzip” entstandene Einführung findet sich zum Beispiel hier:

<https://de.wikibooks.org/wiki/C++-Programmierung>

Ein vollständige Beschreibung von C++ findet sich hier:

<https://en.cppreference.com>

Bei spezifischen Fragen zu C++ (und diversen anderen Programmiersprachen) kann man auf dieser Seite suchen:

<https://stackoverflow.com/>

Dort sind sehr viele Fragen bereits gestellt und beantwortet worden.

2 Einführung

2.1 Hello World

Als erstes Beispiel für eine neue Programmiersprache wird traditionell ein Programm gezeigt, das nichts weiter tut als den Text „Hello World“ auf dem Computerbildschirm erscheinen zu lassen. Ein solches wollen auch wir in Listing 1 angeben.

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

Listing 1: Ein Hallo-Welt-Programm in C++

Wir können an dieser Stelle noch nicht genau auf die Bedeutung aller Programmierbefehle eingehen, wollen aber dennoch alles kommentieren. Die erste Zeile sorgt dafür, dass unserem Programm die Befehle zur Verfügung stehen, um Text auszugeben. Die nächste Zeile `int main()` markiert den Einstiegspunkt des Programms, d.h. die Stelle, ab der beim Start später mit der Ausführung begonnen werden soll. Jedes vollständige C++-Programm muss eine solche *main*-Funktion haben. Die auszuführenden Befehle sind in einem sogenannten *Block* zusammengefasst, welcher mit geschweiften Klammern umschlossen ist. Die Befehle selbst sind überschaubar: Der erste erzeugt die Ausgabe von „Hello World“ und der zweite beendet das Programm. Dabei wird der sogenannte *Fehlercode* 0 zurückgegeben, welcher signalisiert, dass beim Ausführen des Programms kein Fehler aufgetreten ist. Dieser Rückgabewert ist für den Anwender des Programms später nicht erkennbar: er kann jedoch dazu dienen, verschiedene Programme miteinander kommunizieren zu lassen.

Außerdem bemerken wir an dieser Stelle, dass in C++ jeder *Befehl durch ein Semikolon beendet werden muss*. Dies ist eine wichtige Regel, deren Missachtung häufig zu scheinbar unerklärlichen Fehlern bei der Kompilierung führt. In der Tat dienen die Zeilenumbrüche im Quellcode „nur“ der Übersichtlichkeit, ein Befehl wird durch das abschließende Semikolon beendet. Daher wäre der folgende Quellcode in Listing 2 zum obigen äquivalent und absolut korrekt:

```
1 #include <iostream>
2 int main() { std::cout << "Hello World" << std::endl; return 0; }
```

Listing 2: Ein Hallo-Welt-Programm in zwei Zeilen

2.2 Der Speicher

Wenn wir von Speicher sprechen, so meinen wir nicht die Festplatte, sondern ein Bauteil des Computers, das während des laufenden Betriebs Daten nur für die Dauer eines Programmablaufs abspeichert. Man bezeichnet dies auch als RAM (Random Access Memory).

Der Speicher ist eine durchnummerierte Aneinanderreihung von Speicherzellen. Eine Speicherzelle ist ein elektronischer Chip, welcher wiederum 8 Bauteile enthält: Diese Bauteile nennt man *Bits*. Ein Bit kann geladen und entladen werden, hat somit immer genau einen Zustand 1 oder 0. Jede Speicherzelle kann daher $2^8 = 256$ Zustände annehmen (mögliche Kombinationen von Zuständen der einzelnen 8 Bits). Man

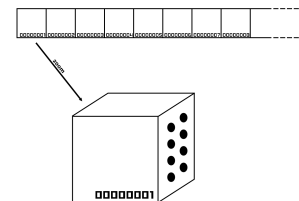


Abbildung 1: Der Speicher

kann diese Zustände also als ganze Zahlen zwischen 0 und 255 interpretieren. Diese Interpretation ist gegeben durch die Darstellung einer Zahl im Binärformat. Eine Speicherzelle bezeichnet man auch als *Byte*. Die Speicherzelle hat 8 ausgehende Drähte, auf welchen nur Strom fließt, wenn das dazugehörige Bit gesetzt (also 1) ist. Aus technischen Gründen kann immer nur ein ganzes Byte auf einmal gelesen oder neu beschrieben werden, keine einzelnen Bits.

Man möchte auch negative Zahlen in Bytes codieren können. Man könnte dafür das erste Bit als sogenanntes *Vorzeichenbit* reservieren, um sich zu merken, ob die Zahl positiv (Vorzeichenbit gleich 0) oder negativ (Vorzeichenbit gleich 1) ist. Die restlichen Bits können dann nur noch 128 verschiedene Zustände annehmen, also können wir nun die Zahlen von -127 bis 127 darstellen. Dieses Prinzip zeigt anschaulich, dass es einen markanten Unterschied zwischen Daten und deren Interpretation gibt. Ein Byte kann als positive Zahl zwischen 0 und 255 oder aber als vorzeichenbehaftete Zahl zwischen -127 und 127 interpretiert werden. Beides verwendet jedoch das gleiche Speichermedium. Man bezeichnet eine solche Interpretation als *Datentyp*. In der Realität wird zur Darstellung negativer Zahlen ein anderes Format, genannt „Zweierkomplement“, verwendet, welches praktischer zu implementieren ist und nur eine Null enthält (das obige Format hat eine $+0$ und eine -0). Daher kann man mit dem Zweierkomplement mit einem Byte Zahlen von -128 bis 127 darstellen. Der Inhalt eines Bytes kann aber auch als ein Buchstabe oder ein anderes Zeichen interpretiert werden (beispielsweise über den ASCII-Code).

Durch Zusammenschluss von Speicherzellen lassen sich auch größere Zahlen darstellen. Den Zusammenschluss von zwei Bytes bezeichnet man als *Word* (Wort), es kann bereits $2^{16} = 65536$ Zustände annehmen. Ein *DWord* (Doppelwort) ist der Zusammenschluss von zwei Words und daher 4 Bytes oder 32 Bit lang. Es kann zum Speichern von Zahlen zwischen 0 und $2^{32} - 1 = 4294967295$ verwendet werden. Dementsprechend bezeichnet man 64-Bit-Speicherblöcke als *QWord* (Quad Word).

Eine Variable, die nur ein einzelnes Byte umfasst, wird gelegentlich auch als *char* bezeichnet, für „Character“. Der Name dieses Datentyps leitet sich daraus her, dass einzelne Buchstaben und andere Zeichen als Zahlen von 0 bis 255 im Computer abgespeichert werden. Zeichenketten und ganze Texte sind somit Speicherblöcke von n aufeinanderfolgenden Bytes (chars), wobei n die Länge der Zeichenkette ist.

Gelegentlich ist es nötig, auch über eine Darstellung reeller Zahlen zu verfügen. Dafür werden meist 8 Bytes Speicher verwendet, die von einem internen Subprozessor als Kommazahlen interpretiert werden. Auf die genaue Realisierung werden wir nicht näher eingehen. Dieser Datentyp trägt den Bezeichner *double*.

2.3 Maschinencode und Kompilierung

Computer wurden ursprünglich als aufwendige Rechenmaschinen entworfen. Sie alle enthalten einen Kernchip, welcher auch heute noch alle tatsächlichen Berechnungen durchführt. Dieser Baustein ist die Central Processing Unit, auch kurz CPU. Die CPU enthält intern eine sehr geringe Anzahl Speicherzellen (etwa 8 bis 30), die jeweils für gewöhnlich 32 oder 64 Bits speichern können. Dies nennt man auch die *Registergröße* oder *Wortgröße* der CPU, die Speicherzellen selbst dementsprechend *Register*.

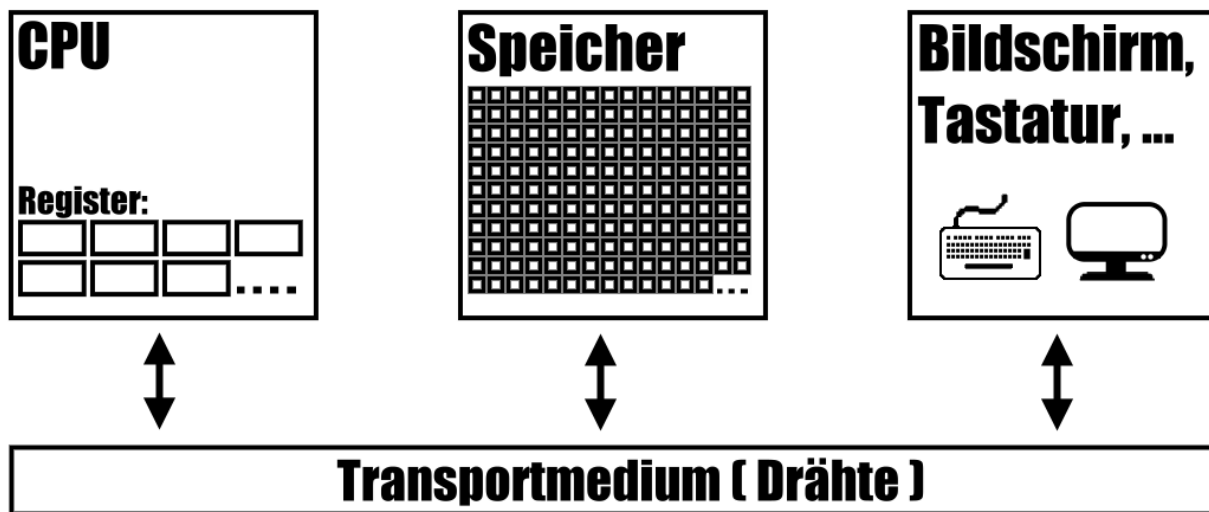


Abbildung 2: Schematischer Aufbau eines Computers

Die CPU eines Computers kann nur eine sehr geringe Anzahl von rudimentären Rechenoperationen durchführen. Genau wollen wir darauf nicht eingehen, doch besteht ein solcher CPU-Befehl beispielsweise daraus, den Inhalt zweier Register zu addieren, subtrahieren, multiplizieren, dividieren oder ähnliche arithmetische Operationen durchzuführen. Natürlich kann die CPU auch bis zu einer Registergröße Daten aus dem Speicher in ein Register laden, oder aus einem Register Daten in den Speicher schreiben. Jedem CPU-Befehl ist ein numerischer Code zugewiesen, welcher in einem Word gespeichert werden kann. Die so codierten CPU-Befehle heißen *Maschinencode*. Um ein Computerprogramm auszuführen, liest die CPU aus dem Speicher Maschinencode ein und führt die Befehle nacheinander aus. Dabei sind auch Sprünge zu Befehlen im Code möglich, und insbesondere kann die Wahl des nächsten Befehls vom Inhalt eines bestimmten Registers abhängen (bedingter Sprung). Es ist nun jedoch ausgesprochen mühsam, auf diese Art und Weise Algorithmen zu implementieren: Dies führte zur Entwicklung von Programmiersprachen, die eine für Menschen wesentlich zugänglichere Syntax vorweisen können. Als *Compiler* bezeichnet man Programme, die den Programmcode einer Programmiersprache in Maschinencode übersetzen. Diesen Vorgang nennt man *Kompilierung*. Der Compiler selbst muss freilich irgendwann mühsam als Maschinencode implementiert worden sein.

2.4 Die Standardbibliothek

Die Standardbibliothek ist ein wichtiger und ungemein nützlicher Bestandteil von C++. Sie stellt eine Fülle von Funktionalitäten zur Verfügung, mit denen sich z.B. Daten verwalten lassen (siehe etwa `std::vector` in Abschnitt 6.2) oder die Verbindung mit der Außenwelt herstellen lässt. Die Module der Standardbibliothek sind bereits kompiliert und können von selbst geschriebenen C++-Programmen benutzt werden (mehr zur Einbindung von vorkompiliertem Code in Abschnitt 5.4). Man erkennt Dinge, die aus der Standardbibliothek kommen daran, dass ihr Name mit `std::` beginnt. Ein Beispiel ist `std::cout` aus dem Hello-World-Programm, das einen Text ausgibt. Ein weiteres Beispiel ist `std::endl`, das dafür sorgt, dass in der Ausgabe eine neue Zeile begonnen wird. Schnittstellen (also Spezifikationen, auf welche Weise Dinge aufgerufen werden

müssen) von Funktionalitäten der Standardbibliothek sind in sogenannten *Headerdateien* zusammengefasst. Diese Headerdateien muss man in das C++-Programm einbinden. In unserem Hello-World-Programm haben wir mit

```
1 #include <iostream>
```

eine Headerdatei eingebunden, die viele Funktionen zum Ausgeben und Einlesen von Daten bereit hält. Es gehören aber noch viele andere Headerdateien zur Standardbibliothek.

2.5 C und C++

C++ wurde 1985 als Erweiterung der Programmiersprache C von Bjarne Stroustrup entwickelt. Da es sich um eine Erweiterung handelt, ist prinzipiell jeder gültige C-Code (wenn er keine Schlüsselwörter benutzt, die es in C++, aber nicht in C gibt) auch gültiger C++-Code. Die Erweiterung bezieht sich insbesondere darauf, dass man in C++ Klassen zur Verfügung hat, mit denen man eigene Objekt erstellen kann (siehe Abschnitt 7). Daneben gibt es eine Vielzahl von syntaktischen Erweiterungen. Auch C++ selbst ist nicht statisch, es gibt alle paar Jahre neue Standards mit neuen Funktionalitäten. Auch hier gilt das Prinzip der Abwärtskompatibilität, d.h. Programme, die einem älteren Standard genügen, genügen auch dem neuen. Der aktuelle Standard ist C++20, und es gibt Vorabversionen von C++23. In diesem Kurs wird aber im Wesentlichen C++11 erklärt. Insbesondere sollten alle Compiler, die für den Standard C++11 geschrieben wurden, die hier vorgestellten Programme kompilieren können.

3 Programmierumgebungen

3.1 Nutzung der PC-Pools

Das Institut für angewandte Mathematik betreibt zwei PC-Pools, einen im Mathematikzentrum (Endenicher Allee 60, Nebengebäude, Raum N0.004 / N0.005) und einen im Rechenzentrum (Wegelerstraße 6, Raum E0.013).

Auf den Rechnern in beiden PC-Pools ist Linux installiert. Für die Bedienung von Linux eignen sich dieselben Befehle, die in Tabelle 1 für die Benutzung von Cygwin angegeben sind. Allgemein überträgt sich die Cygwin-Anleitung aus Abschnitt 3.7 auf die Bedienung von Linux. Auf den Linux-Rechnern in den PC-Pools gibt es aber nicht den unten angegebenen Editor **notepad**, stattdessen (z.B.) die Editoren **geany**, **kate** und **emacs**.

Auf allen Rechnern in den PC-Pools findet sich der Compiler **g++**, sodass Programme dort auch mit dem Befehl

```
g++ -std=c++11 -Wall -Wpedantic -o EXECUTABLE QUELLDATEI
```


kompiliert werden können (s.u. für eine Erläuterung).

3.2 g++ unter Windows

Der Compiler, mit dem wir unser Hello-World-Programm und auch zukünftige Übungen in ausführbaren Maschinencode übersetzen werden, ist der C++-Compiler aus der GNU Compiler Collection, welchen wir hier kurz exemplarisch einführen wollen. Er trägt den Namen *g++*. Obgleich er ein sehr weit verbreiteter und gängiger Compiler ist, ist er selbstverständlich nicht der Weisheit letzter Schluss - es gibt eine Vielzahl weiterer Compiler, von denen einige leider nur käuflich zu erwerben sind.

Der *g++* ist ein unter Linux entwickelter Compiler. Für eine ganze Sammlung von Linux-Programmen existieren Windows-Ports: Diese Sammlung heißt Cygwin. Wir werden hier kurz erläutern, wie Cygwin zu installieren und zu bedienen ist. Außerdem werden wir in der zweiten Woche des Kurses voraussichtlich noch die Eclipse IDE mit den C/C++ Developer Tools und der Cygwin Toolchain verwenden. Zunächst beschreiben wir hier aber, wie man mit Cygwin arbeiten kann, was für den Anfang völlig ausreicht.

3.3 Architektur (32bit oder 64bit?)

Man sollte für die Installation wissen, ob man ein 32- oder 64-Bit-Windows installiert hat. Einigermassen neue Rechnern werden immer 64-Bit-Systeme sein. Wenn man das aber nicht weiß, kann man es nachsehen, wenn man +R drückt und dort `control /name Microsoft.System` eingibt. Dort steht zum Beispiel *“System type: 64-bit Operating System”*.

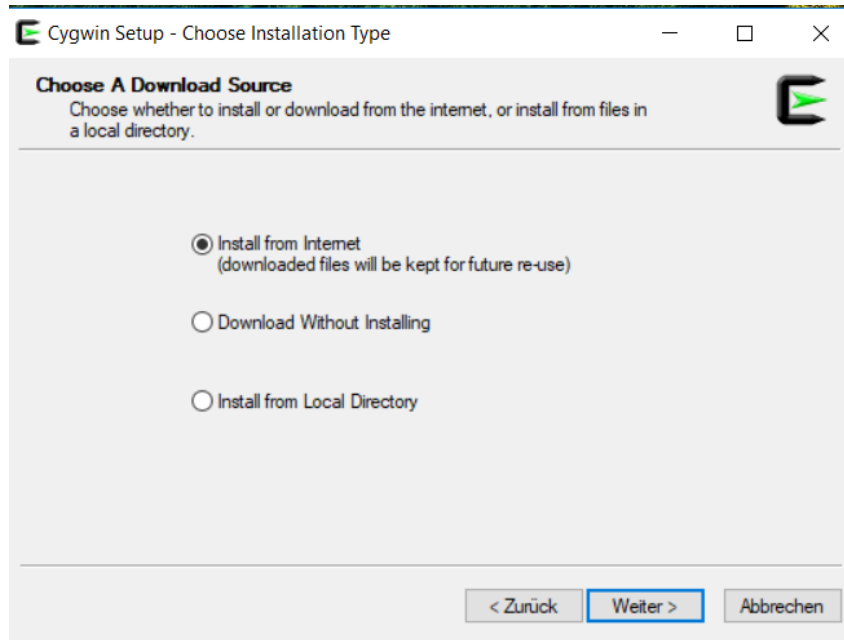
Auf 64-Bit-Betriebssystemen kann man auch 32-Bit-Software benutzen, aber nicht andersherum.

3.4 Cygwin installieren

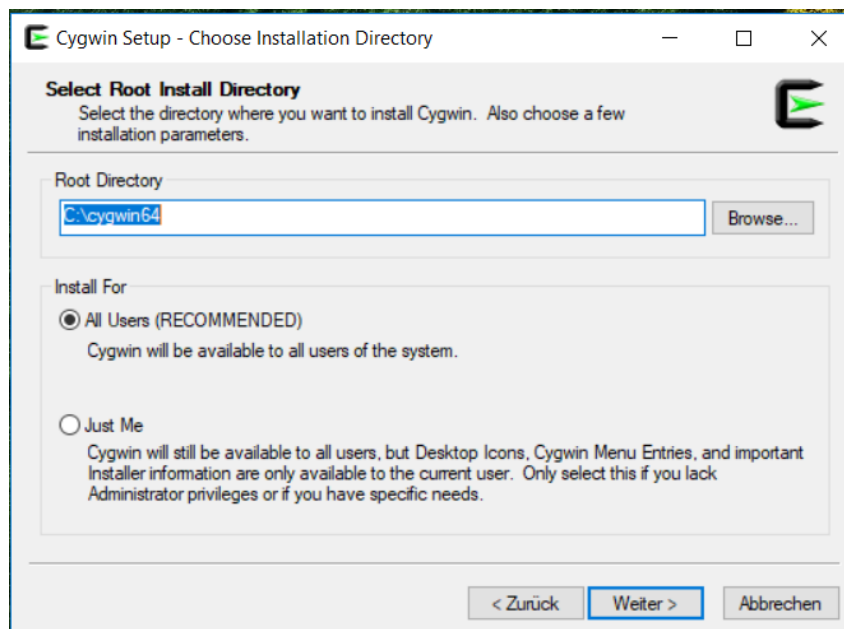
Das Cygwin-Setup kann man auf <http://cygwin.org/> herunterladen.

Das dort heruntergeladene `setup-x86_64.exe` bzw. `setup-x86.exe` lässt man laufen und klickt sich durch die folgenden Fenster.

Man gibt zunächst an, dass man cygwin aus dem Internet heraus installieren will:

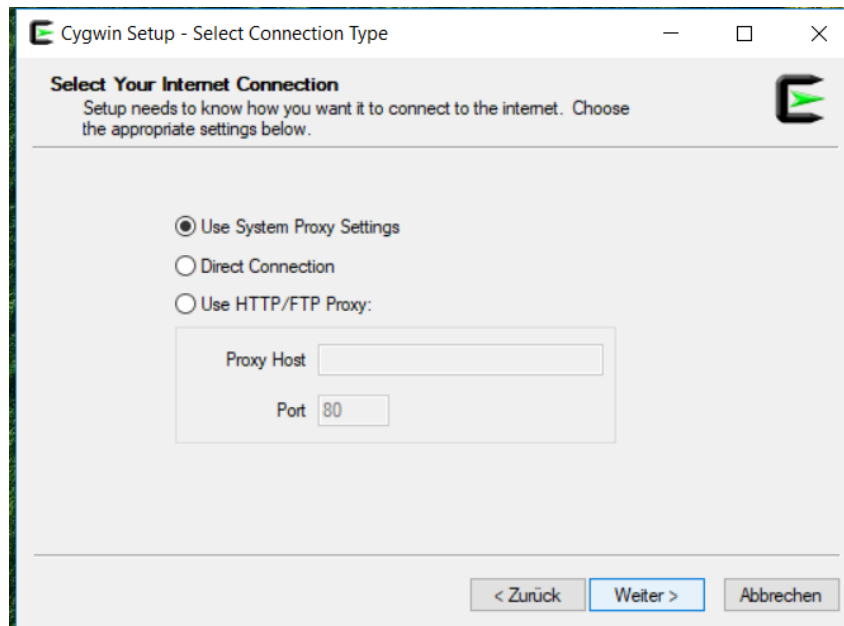


Anschließend wählt man ein Verzeichnis aus. Es ist wichtig, Cygwin in einen Pfad ohne Leerzeichen o.ä. zu installieren. Belassen Sie es also bitte bei dem empfohlenen Installationspfad `C:\cygwin` bzw. `C:\cygwin64`:

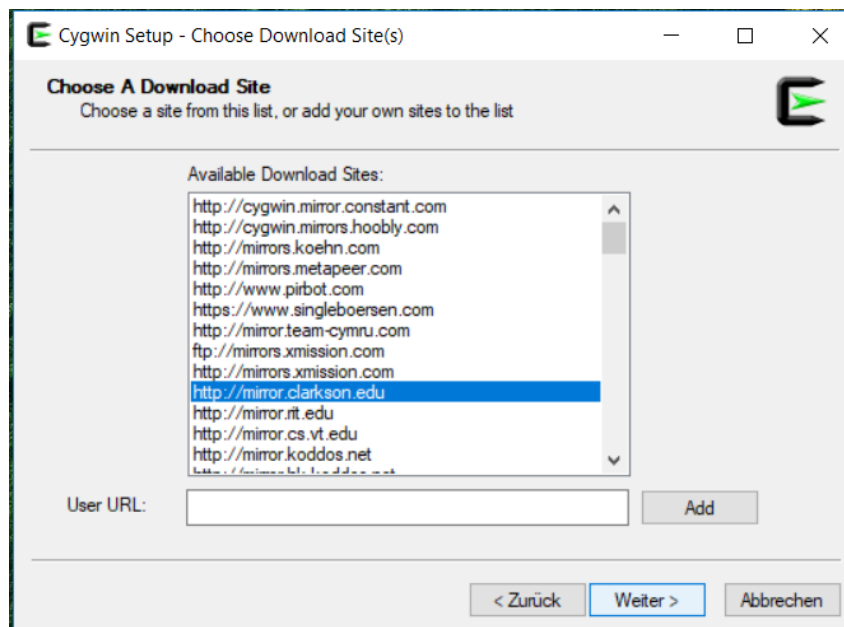


Danach gibt man ein Download-Verzeichnis an, wobei man es bei dem empfohlenen Pfad belassen kann.

Im nächsten Schritt muss man angeben, wie man sich mit dem Internet verbindet. Typischerweise so:



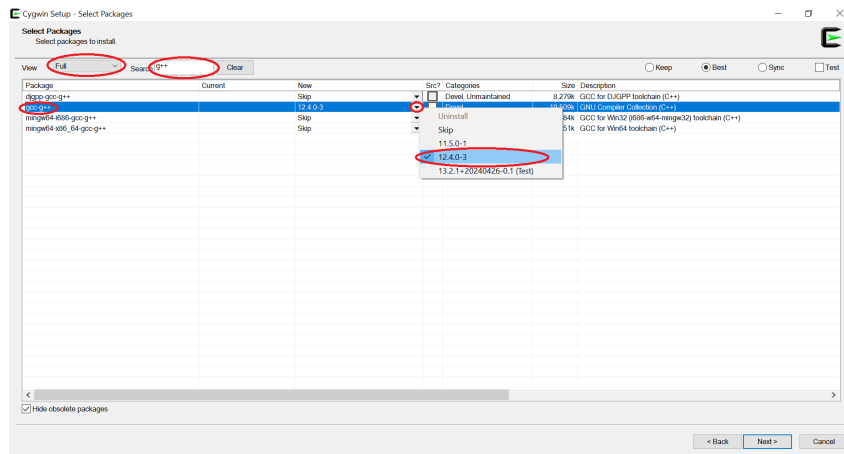
Anschließend kann man wählen, von wo cygwin heruntergeladen werden soll. Hier geht im Prinzip jeder der Vorschläge (wobei natürlich nicht alle Verbindungen gleich schnell sind):



In dem folgenden Fenster kann man nun die benötigten Pakete angeben. Man wählt hier:

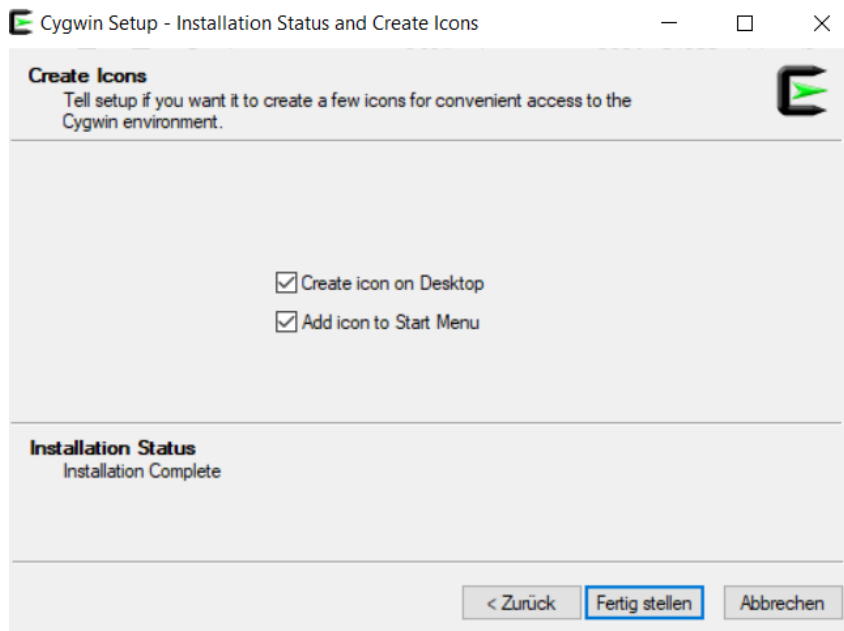
- gcc-core
- g++
- gdb
- make

Diese sucht man (wobei die Auswahl “Full” eingestellt sein sollte) mit der Suchfunktion und klickt sie entsprechend an. In dem folgenden Beispiel sind gleich alle gefundenen g++-Pakete angeklickt (man kann selektiver vorgehen, aber es schadet im Prinzip nichts, sich zu viel herunterzuladen):



Das Programm untersucht selbst, welche Pakete mit der Auswahl noch benötigt werden und listet diese auf, damit man sie bestätigen kann.

Zum Schluss kann man sich noch ein Icon auf dem Desktop wünschen:



Wenn man Pakete nachinstallieren will, kann man `setup-x86_64.exe` bzw. `setup-x86.exe` auch später erneut laufen lassen.

3.5 Eine fertige Cygwin-Version

Eine weitere Möglichkeit, cygwin auf seinem Windows-Rechner laufen zu lassen, besteht darin, sich eine gezippte cygwin-Version von der Veranstaltungsseite

https://precampus.uni-bonn.de/goto.php?target=crs_78&client_id=precampus

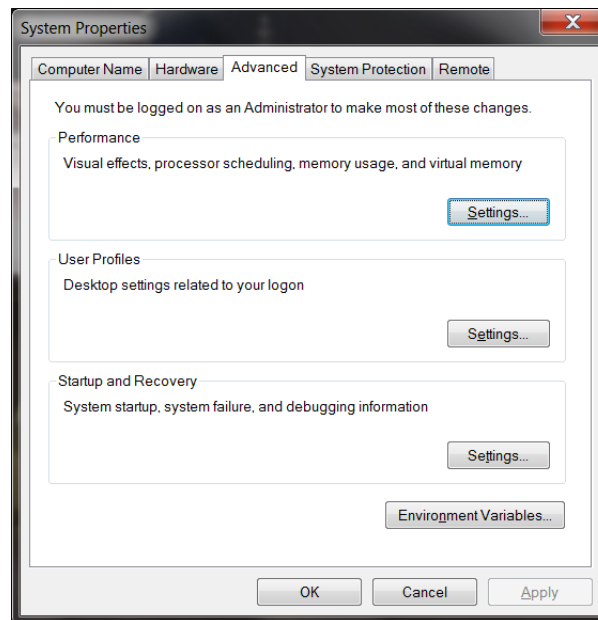
oder (wenn man z.B. noch nicht in PreCampus angemeldet ist) auf der Veranstaltungsseite

www.or.uni-bonn.de/lectures/ws24/programmierkurs_ws24.html

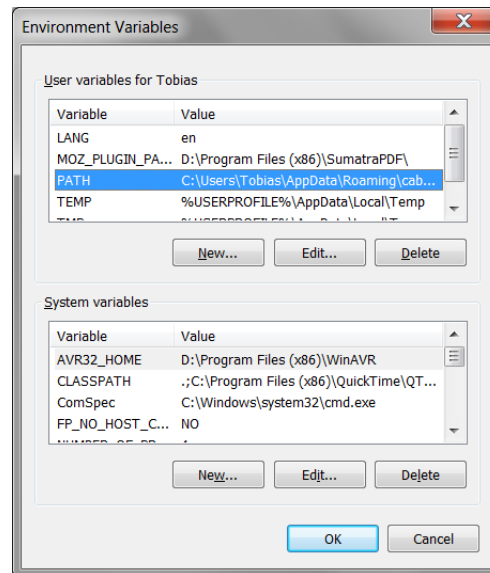
herunterzuladen. Diese gezippte Datei muss man entpacken. Danach findet man in dem entstandenen Verzeichnis eine Datei namens **start_Cygwin.bat**, die man ausführt, um cygwin zu starten. Der Compiler ist dann gleich dabei.

3.6 PATH setzen

Wenn man später mit eclipse arbeiten will, muss man Cygwin zu der Umgebungsvariable PATH hinzufügen, damit eclipse die benötigten Programme findet. Dafür drückt man **Win+R** und gibt **"control sysdm.cpl,,3"** (sic) ein. Es öffnet sich ein Fenster *System Properties*.



Dort klickt man auf **Environment Variables**.



Schauen Sie bitte oben unter *User variables* nach, ob dort bereits eine Variable mit dem Namen `PATH` existiert. Falls nicht, legt man eine neue Variable mit dem Namen `PATH` und dem Wert `"C:\cygwin64\bin"` an (falls man Cygwin in `C:\cygwin64` installiert hat, sonst passt man den Pfad bitte entsprechend an). Falls die Variable bereits existiert, editiert man sie, indem man den Wert `"C:\cygwin64\bin"` als letzten Eintrag hinzufügt.

3.7 Bedienung von Cygwin

Cygwin selbst lässt sich nun vom Startmenü aus aufrufen und präsentiert sich als schwarzes Fenster mit einer blinkenden Eingabe, etwa wie folgt:

```
rattle@lucy ~
$
```

Hinter dem Dollarzeichen erwartet Cygwin nun einen Befehl. Es gibt zahlreiche Befehle, einige wichtige haben wir hier aufgelistet:

Befehl	Effekt
<code>ls</code>	Listet den Inhalt des derzeitigen Verzeichnisses auf.
<code>mkdir <name></code>	Erstellt einen Ordner mit dem angegebenen Namen
<code>cd <ordner></code>	Wechselt in den angegebenen Ordner.
<code>cp <quelle> <ziel></code>	Kopiert die Datei <code>quelle</code> nach <code>ziel</code> .
<code>mv <quelle> <ziel></code>	Verschiebt die Datei <code>quelle</code> nach <code>ziel</code> .
<code>rm <datei></code>	Löscht eine Datei.

Tabelle 1: Befehle der Cygwin-Kommandozeile

Ein einzelner Punkt steht für das derzeitige Verzeichnis und zwei Punkte für das darüberliegende. Der Befehl `cd .` hat also keinen Effekt und `cd ..` bewegt sich einen Ordner nach oben. Darüber hinaus ist jedes Programm, das auf dem Computer (bzw. in Cygwin) installiert ist, ein Befehl. Durch Eingabe von `notepad` beispielsweise öffnet sich der Windows-Texteditor und der Befehl `g++` ruft den Compiler auf. Nun wollen wir unser Hello-World-Programm aus Abschnitt 2.1

kompilieren und ausführen. Dazu gibt man zuerst die folgenden Befehle ein (jeweils durch ein “Enter” beendet).

```
mkdir ckurs
cd ckurs
notepad helloworld.cpp &
```

Mit den ersten beiden Befehlen hat man ein Verzeichnis namens `ckurs` erstellt und ist in dieses gewechselt. Nach `notepad helloworld.cpp` öffnet sich ein Editor-Fenster, in dem man sein Hello-World-Programm (oder jedes andere C++-Programm) abtippen kann. Natürlich muss man es danach abspeichern. Es ist Konvention, dass Dateien, welche C++-Quellcode enthalten, die Dateierstreckung `.cpp` erhalten. Nach Tippen des oben angegebenen Quellcodes speichern wir die Datei und kehren zur Kommandozeile zurück. Der Befehl zum Kompilieren

```
g++ -std=c++11 -Wall -Wpedantic -o EXECUTABLE QUELLDATEI
```

wobei in diesem Fall unsere Quelldatei den Namen `helloworld.cpp` trägt. Als Name für die Executable (mit `-o EXECUTABLE`) angegeben bietet sich der Name `helloworld.exe` an, doch natürlich steht einem die Entscheidung hier frei. Die Option `-Wall` ist eine Abkürzung für „Warning: All“ und bedeutet, dass der Compiler alle Warnungen ausgibt. Warnungen sind unser wichtigstes Hilfsmittel, um später Fehler in Programmen zu finden und zu beheben. Die Option `-Wpedantic` fordert den Compiler dazu auf, sehr kleinlich zu sein (was für einen Compiler immer gut ist).

In unserem Fall kann der Aufruf also z.B. so lauten:

```
g++ -std=c++11 -Wall -Wpedantic -o helloworld.exe helloworld.cpp
```

Nachdem wir den `g++` aufgerufen haben, wurde im gleichen Verzeichnis eine Datei erstellt, die `helloworld.exe` heißt. Wenn das Kompilieren funktioniert hat, kann man anschließend das Programm mit Eingabe von

```
./helloworld.exe
```

(gefolgt von “Enter”) ausführen. Der Befehl `./helloworld.exe` besagt, dass die Datei `helloworld.exe` im derzeitigen Verzeichnis (der einzelne Punkt) ausgeführt werden soll.

Man kann den Parameter `-o EXECUTABLE` in obigem Aufruf weglassen, dann wird eine ausführbare Datei namens `a.exe` (bzw. in der Linux-Welt `a.out`) erzeugt.

Beachten Sie, dass die Verzeichnisse, die in Cygwin verwendet werden, in dem (aus Sicht von Windows) globalen Verzeichnis stehen, in dem Sie Cygwin installiert haben (und dort in einem Unterverzeichnis namens `home`).

3.8 Installation unter Mac OS X

Um C/C++-Code unter Mac OS X zu schreiben, muss ein Editor installiert werden. Als Editor kann man z.B. Atom.io (<https://atom.io>) nutzen.

Als Compiler bietet sich der `g++`-Compiler an (also derselbe Compiler, der auch in Cygwin verwendet wird). Der aktuelle `g++` Compiler wird auf der Apple-Website im Package Xcode-Command-Line-Tools (<https://developer.apple.com/download/more/?=command%20line%20tools>) zum Download angeboten. Um über den Link in das Developer-Portal von

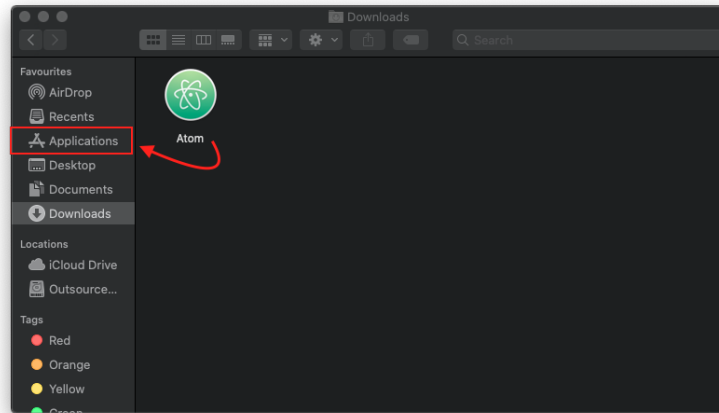


Abbildung 3: Installation von Atom: Nach dem Download von Atom reicht ein Verschieben der App in den Programmordner aus, um die App zu installieren.

Apple zu gelangen, muss man sich mit einer *Apple ID* anmelden. Danach lässt sich die aktuelle (nicht-beta) Version der Command-Line-Tools in dem Portal herunterladen.

Die Installation verläuft wie eine reguläre Setup-Routine.

Nach der Installation kann der Terminal (Programme -i, Dienstprogramme) C/ C++ Code kompilieren.

3.9 Bedienen von g++ unter Mac OS X

Es bietet sich an für die Programmieraufgaben ein separates Arbeitsverzeichnis zu erstellen. Am besten sind innerhalb dieses Ordners für jede Programmieraufgabe (oder zumindest für jeden Tag) separate Ordner, um die Übersicht zu behalten.

Nun muss der Terminal noch in das Verzeichnis geleitet werden, in dem gearbeitet werden soll (siehe Tabelle in 3.7: Terminalbefehl `cd +Pfad`).

Neue C++ Dateien können mit dem Terminal mit `touch dateiname.cpp` erstellt werden.

Ist der C++ Code nun fertig geschrieben, kann er in Atom abgespeichert und vom Terminal kompiliert werden. Wichtig ist, dass sich der Terminal in demselben Verzeichnis befindet, wie die zu kompilierende Datei. Kompiliert wird mit: `g++ -Wall -Wpedantic -o programmname quelltext.cpp`

Wenn alles richtig kompiliert wurde, erscheint in dem betrachteten Verzeichnis eine ausführbare *Unix Executable* Datei. Diese lässt sich über `./programmname` direkt im Terminal ausführen.

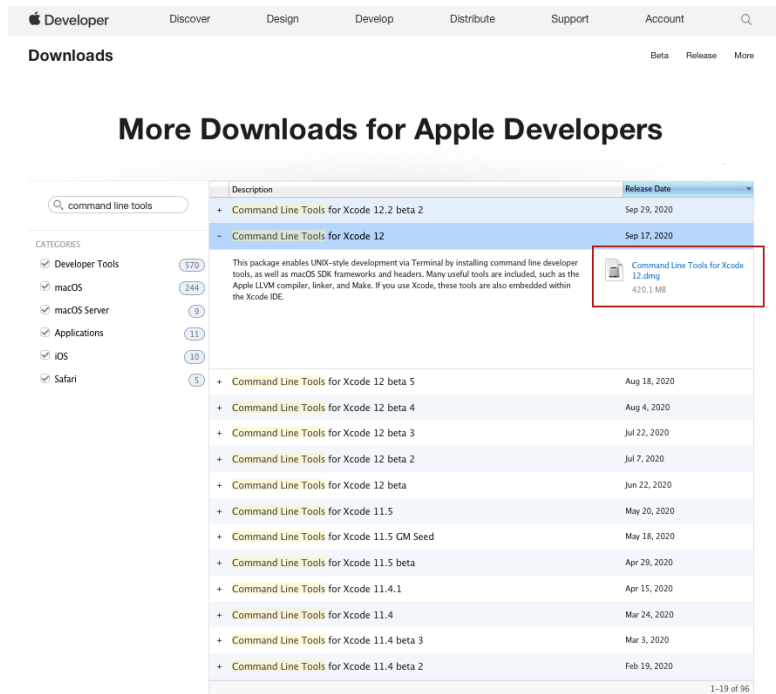


Abbildung 4: Download von den Command Line Tools

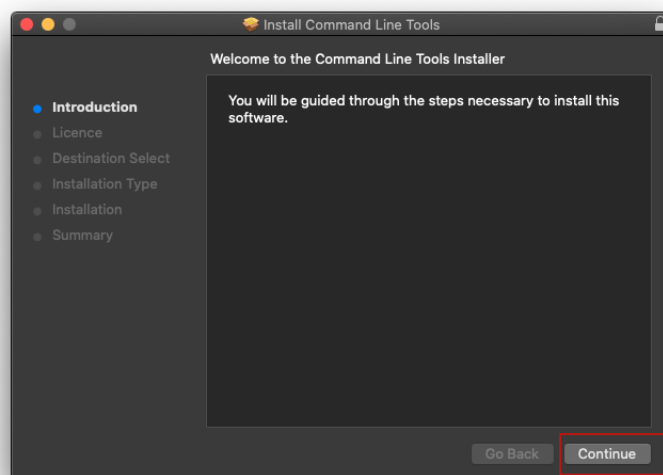


Abbildung 5: Setup der Command Line Tools: Für Installation immer auf *weiter* klicken.

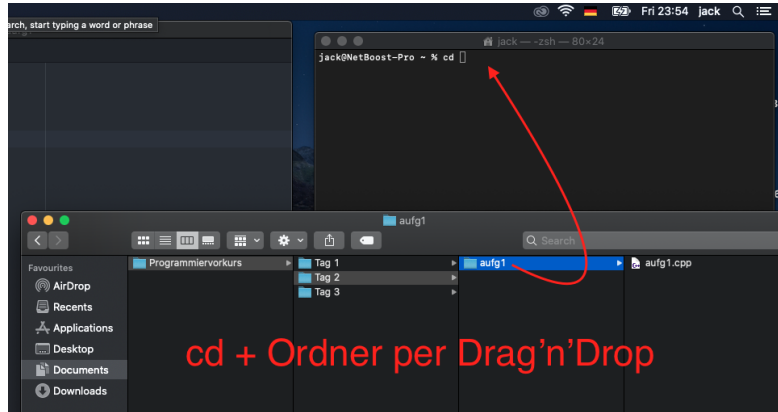


Abbildung 6: Wer schneller Pfade im Terminal wechseln möchte, kann auch `cd` tippen und das Verzeichnis vom Finder per Drag'n'Drop in den Terminal ziehen.

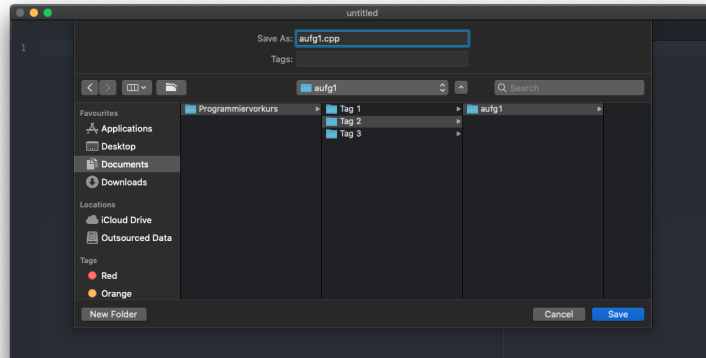


Abbildung 7: Alternative Erstellung von C++ Dateien: Wird über Atom eine Datei mit Dateiendung `.cpp` abgespeichert, wird sie direkt als C++ Datei markiert. Atom färbt den Code anschließend automatisch im C++ Syntax farblich ein.

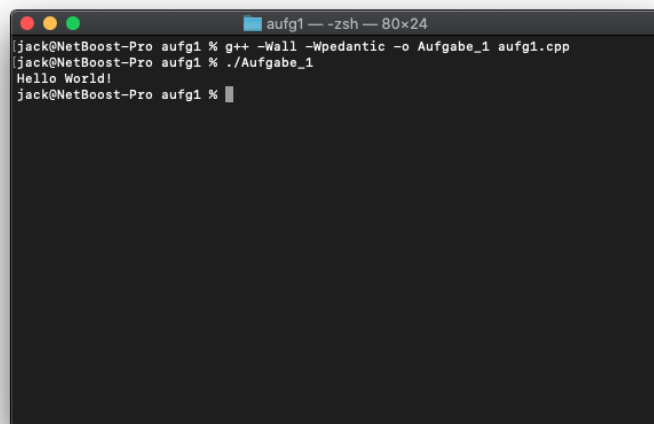
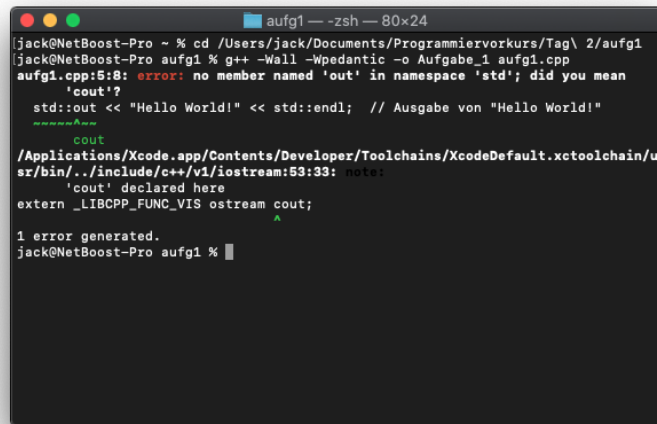


Abbildung 8: Erfolgreiches Kompilieren und Ausführen von *Hello World!*

A terminal window titled 'aufg1 — zsh — 80x24' showing a C++ compilation process. The user runs 'g++ -Wall -Wpedantic -o Aufgabe_1 aufg1.cpp'. The compiler reports an error: 'error: no member named 'out' in namespace 'std'; did you mean 'cout'?'. The code snippet shows 'std::out << "Hello World!" << std::endl; // Ausgabe von "Hello World!"'. Below the error, the compiler notes that 'cout' is declared in the C++ standard library and provides the path to the header file. The terminal concludes with '1 error generated.' and the prompt 'jack@NetBoost-Pro aufg1 %'.

```
jack@NetBoost-Pro ~ % cd /Users/jack/Documents/Programmivorkurs/Tag\ 2/aufg1 ]
jack@NetBoost-Pro aufg1 % g++ -Wall -Wpedantic -o Aufgabe_1 aufg1.cpp ]
aufg1.cpp:5:8: error: no member named 'out' in namespace 'std'; did you mean
      'cout'?
      std::out << "Hello World!" << std::endl; // Ausgabe von "Hello World!"
      ~~~~~^
      cout
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/u
sr/bin/../include/c++/v1/iostream:53:33: note:
      'cout' declared here
extern _LIBCPP_FUNC_VIS ostream cout;
                                ^
1 error generated.
jack@NetBoost-Pro aufg1 %
```

Abbildung 9: War das Kompilieren nicht erfolgreich, versucht der Terminal zu helfen.

4 Elementare Sprachkonstrukte

4.1 Kommentare

Obleich Programmiersprachen gedacht sind, um dem Menschen verständlicher zu sein als der kryptische Maschinencode, können nur die wenigsten von uns C++-Quellcode wie ein lustiges Taschenbuch lesen. Daher möchte man häufig an verschiedenen Stellen im Quellcode sogenannte *Kommentare* einfügen, d.h. Erläuterungen und Erklärungen zum Programm, welche nicht vom Compiler als Befehle interpretiert werden sollen. Es gibt in C++ zwei Arten, Kommentare einzufügen. Zum einen kann man die Zeichenfolge `//` verwenden. Alles was in einer Zeile nach `//` steht, wird als Kommentar interpretiert. Zum anderen kann man einen Kommentar mit der Zeichenfolge `/*` beginnen und mit der Zeichenfolge `*/` beenden. Hierdurch lassen sich auch mehrzeilige Kommentare einfügen. Listing 3 ist ein Beispiel.

```
1  /* Ein Programm, das den
2  Text "Hello World" ausgibt. */
3  #include <iostream>
4
5  int main()
6  {
7      std::cout << "Hello World" << std::endl; // Hier wird der Text ausgegeben.
8      return 0;                               // Programm wird fehlerfrei beendet.
9  }
```

Listing 3: Ein Hallo-Welt-Programm mit (recht sinnlosen) Kommentaren

Dieses kleine Beispiel wäre natürlich auch ohne Kommentare leicht zu verstehen. Aber gerade in größeren und komplizierteren Programmen sind Kommentare sehr nützlich.

4.2 Variablen

Ganz abstrakt ist ein Programm eine Maschinerie, die gewisse Daten erhält, und daraus neue Daten auf eine bestimmte Art und Weise berechnet. Daten treten in einem Programm stets in Form von sogenannten *Variablen* auf. Dabei ist eine Variable der Name für eine zusammenhängenden Region im Speicher des Computers, die durch ihren Datentyp eine Interpretation der dort gespeicherten Bits zugewiesen bekommt. Durch den Namen lässt sich im C++-Programm die Speicherregion auslesen oder neu beschreiben. Der Programmierer kann sich zu Beginn eines Programmblocks wie folgt Variablen deklarieren (erstellen):

DATENTYP NAME = WERT;

Wann immer wir Definitionen wie oben angeben, so bedeutet ein unterstrichenes Wort, dass an dieser Stelle verschiedenes stehen kann. Für NAME etwa wird der *Name* eingefügt, welchen die Variable haben soll. Dies ist eine beliebige Zeichenfolge aus Buchstaben, Ziffern und Unterstrichen, welche nicht mit einer Ziffer beginnt. Der Name der Variablen sollte Aufschluss über ihren Zweck im Programm liefern. Variablenamen mit nur einem Buchstaben, obgleich in der Mathematik sehr verbreitet, sorgen bei Programmen in den meisten Fällen nur für Verwirrung. Ist ein Teil einer Definition grau gefärbt, so ist dieser Teil optional. Wir bemerken, dass das Semikolon oben nicht mehr optional ist.

Zur Speicherung ganzer Zahlen bietet sich in den meisten Fällen der Typ `int` an, also z.B.:

```
1  int x = -37;
2  int y;
3  int z = 1984;
```

Will man rationale Zahlen speichern, die nicht notwendigerweise ganze Zahlen sind, wird man typischerweise auf den Typ `double` zurückgreifen:

```
1 double x = 3.72;
```

Hier ist bei Definition auch eine wissenschaftliche Notation möglich, die im folgenden Beispiel die Variable `x` mit $-3,72567 \cdot 10^4 = -37256,7$ initialisiert:

```
1 double x = -3.72567e4;
```

Ein weiterer sehr nützlicher Datentyp ist `bool`. Variablen vom Typ `bool` können nur die Werte 0 und 1 annehmen. Dabei wird die 0 als “falsch” und 1 als “wahr” interpretiert. Man kann Variablen vom Typ `bool` also wie folgt initialisieren:

```
1 bool a = 0;
2 bool b = 1;
```

oder (mit dem gleichen Effekt):

```
1 bool a = false;
2 bool b = true;
```

Tabelle 2 gibt Aufschluss über zur Verfügung stehende *Datentypen*, welche für `DATENTYP` eingesetzt werden können. Ein `int` beansprucht stets weniger oder genauso viel Speicher wie ein `long int` und stets mehr oder genauso viel Speicher wie ein `short int`. Die gewöhnliche Größe in Bits, die ein `int` belegt, hat sich im Laufe der Jahrzehnte von 16 über 32 zu mittlerweile 64 Bits gesteigert und könnte sich in der Zukunft weiter ändern. Insbesondere kann man als Programmierer nicht wissen, welche Größe z.B. in `int` hat. Ähnliches gilt für andere Datentypen. Die `unsigned`-Varianten von `int` können nur nichtnegative Zahlen abspeichern. Dafür kann man etwas größere Zahlen abspeichern, da man ein Bit mehr für die Zahldarstellung benutzen kann. Außerdem kann es nützlich sein, einer Variable gleich am Typ ansehen zu können, dass sie keinen negativen Wert annehmen kann.

Bei einem Typ, dessen Name zusammengesetzt ist und auf `int` endet (z.B. `unsigned int`) kann `int` selbst weggelassen werden, etwa so:

```
1 unsigned x;
```

Für nicht ganzzahlige Zahlen bietet Tabelle 2 die Datentypen `float`, `double` und `long double`. Diese Datentypen speichern Zahlen x , indem sie in der Darstellung

$$x = (-1)^a \cdot b \cdot 2^c$$

(mit $a \in \{0, 1\}$, $b \in [1, \dots, 2)$ und $c \in \mathbb{Z}$) die Zahlen a, b und c binär abspeichern. Die Details dieser Darstellungsart wollen wir hier nicht erörtern. Typische Größen für diese Datentypen sind 32 Bit (`float`), 64 Bit (`double`) und 80 Bit (`long double`), wobei mehr Bits bedeuten, dass man betragsmäßig größere Zahlen darstellen kann (weil mehr Bits für c zur Verfügung stehen) und eine größere Genauigkeit in der Darstellung hat (weil mehr Bits für b zur Verfügung stehen). In den meisten Fällen wird man `double` als Standarddatentyp wählen.

Optional kann einer Variablen bereits bei der Deklaration ein *Wert* zugewiesen werden. Diesen Vorgang bezeichnet man als *Initialisierung* der Variablen. Beispiel:

```
1 int pi = 3; /* pi wird zu 3 initialisiert */
2 int long x; /* x ist undefiniert */
3 int long y = pi; /* y wird zu pi initialisiert */
```

Variablentyp	Deklaration
Ganzzahl	<code>int v;</code>
Kleine Ganzzahl	<code>short int v;</code>
Große Ganzzahl	<code>long int v;</code>
Sehr große Ganzzahl	<code>long long int v;</code>
Ganzzahl ≥ 0	<code>unsigned int v;</code>
Kleine Ganzzahl ≥ 0	<code>unsigned short int v;</code>
Große Ganzzahl ≥ 0	<code>unsigned long int v;</code>
Sehr große Ganzzahl ≥ 0	<code>unsigned long long int v;</code>
Byte (8 Bit)	<code>char c;</code>
Kleine Fließkommazahl	<code>float f;</code>
Große Fließkommazahl	<code>double d;</code>
Sehr große Fließkommazahl	<code>long double l;</code>
Boolesche Variable	<code>bool a;</code>

Tabelle 2: Datentypen

Achtung: Wird eine Variable nicht initialisiert, so ist sie *undefiniert*: Es ist unvorhersehbar, welchen Wert sie hat. Will man mehrere Variablen vom gleichen Typ deklarieren, so ist dies auch möglich, indem man sie nach Angabe des Datentyps lediglich durch Kommata trennt. Damit ist

```
1 int pi = 3, x, y = pi;
```

eine Kurzschreibweise für den Quellcode oben.

Die Werte von Variablen der in Tabelle 2 aufgelisteten Daten können mit `std::cout` ausgegeben werden. Beispielsweise kann der Wert einer Variable `a` vom Typ `int` mit

```
1 std::cout << a;
```

ausgegeben werden. Dabei können auch mehrere Variablen und Texte (aneinandergehängt durch `<<`) mit einem Aufruf von `std::cout` ausgegeben werden:

```
1 #include <iostream>
2
3 int main()
4 {
5     int    a = 37;
6     double b = 42;
7     double c = 3.14;
8     bool   d = true;
9
10    std::cout << "Der Wert von a ist " << a << " und der von b ist " << b << "." << std::endl;
11    std::cout << "Der Wert von c ist " << c << " und der von d ist " << d << "." << std::endl;
12
13    return 0;
14 }
```

Listing 4: Ausgabe von Werten von Variablen

Die Zeichenkette "`std::endl`" bewirkt, dass in der Ausgabe eine neue Zeile begonnen wird. Dieser Code erzeugt daher die folgende Ausgabe

Der Wert von a ist 37 und der von b ist 42.

Der Wert von c ist 3.14 und der von d ist 1.

Man beachte, dass der Wert einer Variable vom Typ `bool` (in diesem Fall der Variable `d`) als Zahl (0 oder 1) interpretiert wird und entsprechend ausgegeben wird.

4.3 Operatoren und Ausdrücke

Ein *Ausdruck* (*Expression*) in C++ steht für einen Teil des Codes, welcher, ganz anschaulich ausgedrückt, einen Wert hat. Eine Variable ist beispielsweise bereits eine Expression, genau wie Konstanten.

Alle anderen Expressions in C++ entstehen aus Konstanten und Variablen durch deren Verknüpfung mittels *Operatoren* und Klammerung. Abstrakt ausgedrückt ordnet ein Operator einem oder mehreren Werten einen neuen Wert zu. So sind etwa alle Grundrechenarten

Operator	Expression	Wert der Expression
Addition	<code>a + b</code>	Summe von <code>a</code> und <code>b</code>
Subtraktion	<code>a - b</code>	Differenz von <code>a</code> und <code>b</code>
Multiplikation	<code>a * b</code>	Produkt von <code>a</code> und <code>b</code>
Division	<code>a / b</code>	Quotient von <code>a</code> und <code>b</code>
Modulo	<code>a % b</code>	Rest einer Ganzzahldivision von <code>a</code> durch <code>b</code>

Tabelle 3: Arithmetische Operatoren

sogenannte *binäre* Operatoren (da sie zwei Werten einen Neuen zuweisen, nämlich gerade das Rechenergebnis). Beispiele für Expressions sind `3+5*9` und `(3+5)*9`. Dabei gilt wie gewohnt: “Punkt- vor Strichrechnung”. Der Wert der Expression ist dann natürlich das Gesamtergebnis (beim ersten Beispiel also 48 und beim Zweiten 72). Wir werden im Laufe des Kurses außer den Grundrechenarten noch viele weitere Operatoren kennen lernen. Der Wert einer Expression kann durch den *Zuweisungsoperator* “`=`” in einer Variablen gespeichert werden:

```
1 pi = (pi+5)*9; /* setzt die Variable pi auf (pi+5)*9 */
```

Der Zuweisungsoperator entspricht also nicht dem mathematischen Gleichheitszeichen, sondern wird gelesen als “wird gesetzt auf”. Wer sich nun fragt, warum dies ein Operator sein soll, sei gesagt, dass eine Zuweisung in C++ auch einen Wert hat, nämlich gerade den Wert, der zugewiesen wird. Damit ist folgender Code korrekt:

```
1 x = pi = x+5*9; /* entspricht x = (pi=x+5*9); */
```

Hier wird also zunächst der Wert von `(x+45)` in der Variablen `pi` gespeichert – das Ergebnis dieser Zuweisungsoperation ist wiederum `(x+45)`, welches dann nach `x` geschrieben wird. Man sagt auch, der Zuweisungsoperator hat einen *Nebeneffekt*, da er nicht nur einen Wert zurückgibt, sondern in Folge seiner Auswertung auch den Inhalt einer Speicherzelle verändert. Da jede Expression einen Wert hat, hat sie auch einen Datentyp. Gelegentlich möchte man durch Operatoren auch Expressions verknüpfen, die formal unterschiedliche Datentypen haben – in diesem Fall muss eine der Expressions in eine Expression vom anderen Typ konvertiert werden. Diesen Vorgang nennt man *Typenumwandlung* oder *Casting*. In vielen Fällen, wie etwa der Verknüpfung zweier Expressions mit Ganzzahltypen, nimmt C++ diese Konvertierung automatisch und meistens auch so vor, wie man es sich wünscht. Möchte man dennoch manuell eine Typkonvertierung durchführen, so geschieht dies durch folgende Syntax:

```
static_cast<DATENTYP>(EXPRESSION);
```

Als Beispiel könne man etwa eine Fließkommazahl in eine Ganzzahl konvertieren:

```
1 #include<iostream>
2
3 int main()
4 {
5     double x = 12.7;
6     int y;
7     y = static_cast<int>(x);
8
9     std::cout << "x: " << x << " y: " << y << std::endl;
10
11     x = -3.8;
12     y = static_cast<int>(x);
13
14     std::cout << "x: " << x << " y: " << y << std::endl;
15 }
```

Listing 5: Umwandlung von Fließkommazahlen in Ganzzahlen

Die Konvertierung von Fließkommazahlen in Ganzzahlen geschieht durch Runden in Richtung 0. Die Ausgabe der obigen Programms sieht daher so aus:

```
x: 12.7 y: 12
x: -3.8 y: -3
```

All dies wirft ein neues Licht auf die oben vorgestellten Rechenoperationen: Diese haben nämlich, abhängig vom Typ ihrer Argumente, eine unterschiedliche Arbeitsweise.

Dividieren wir zwei Ganzzahlen, so wird eine Ganzzahldivision durchgeführt und der dabei entstehende Rest verworfen; also ergibt $1/2$ den Wert 0 und $7/3$ hätte den Wert 2. Durch explizites Typecasting lässt sich hier ein anderes Verhalten erzwingen:

```
1 unsigned x = 1, y = 2;
2 double half = static_cast<double>(x)/y; /* nun hat half den Wert 0.5 */
```

Man beachte, dass hier nur die Variable `x` zu einem `double` gecasted wurde. Bei der folgenden Division durch `y` wird dieses implizit in ein `double` umgewandelt. Denn: Dividiert man eine Ganzzahl durch eine Fließkommazahl oder umgekehrt, so wird die Ganzzahl konvertiert und man erhält das (mehr oder minder) korrekte Ergebnis der Rechnung als Fließkommazahl. Denselben Effekt hätte man also mit

```
1 double half = x/static_cast<double>(y);
```

oder

```
1 double half = static_cast<double>(x)/static_cast<double>(y);
```

erreichen können. Generell gilt: Verknüpfen wir eine Fließkommazahl mit einer Ganzzahl, so wird diese in eine Fließkommazahl konvertiert, und das Ergebnis ist ebenfalls eine Fließkommazahl. Es gibt nun noch einen weiteren nützlichen Rechenoperator, der bei einer Ganzzahldivision das Ergebnis verwirft und statt dessen den Rest als Ergebnis liefert: Der sogenannte *Modulo*-Operator, `%` (ein Prozentzeichen). So wäre etwa $(7\%5)$ eine Expression mit dem Wert 2. Dieser Operator funktioniert nur mit Ganzzahlen.

Häufig hat man in der Programmierung Zuweisungen der Form $a = a \times b$, wobei \times einer der bisherigen, binären Rechenoperatoren ist. Dafür gibt es die Kurzschreibweise $a \times= b$. Ein Beispiel: $a += 1$ würde den Wert von a um 1 erhöhen. Die Situation, eine Variable um 1 zu de- oder inkrementieren, ergibt sich sehr häufig. Dafür verwendet man die folgenden unären Operatoren.

Operator	Art	Wirkung	Wert der Expression
a++	postfix	inkrementiere a	a
++a	präfix	inkrementiere a	a+1
a--	postfix	dekrementiere a	a
--a	präfix	dekrementiere a	a-1

Tabelle 4: Kurzschreibweisen

Anmerkung: Es gibt Expressions, welche aufgrund ihrer Nebeneffekte nicht eindeutig sind, etwa `i=i+++i`. Diese Expression ist syntaktisch korrekt, doch es gibt keinen offiziellen Standard für ihren Wert. Man bezeichnet solche Expressions als *undefiniert*. Jeder Compiler hat bei derartigen Situationen das Recht, über die weitere Verfahrensweise zu entscheiden (Er könnte etwa die Expression auf eine mögliche Art und Weise auswerten oder einen Fehler erzeugen). Man sollte solche Expressions tunlichst vermeiden.

4.4 Dateneingabe über die Konsole

Die Standardbibliothek stellt für die Eingabe von Daten über die Konsole die Funktion `std::cin` zur Verfügung.

```

1 // addition.cpp (Addiere zwei gegebene ganze Zahlen)
2
3 #include <iostream>
4
5 int main()
6 {
7     std::cout << "Bitte die erste ganze Zahl eingeben: ";
8     int a;
9     std::cin >> a; //Eine eingegebene Zahl wird in der Variablen a gespeichert.
10
11     std::cout << "Bitte die zweite ganze Zahl eingeben: ";
12     int b;
13     std::cin >> b; //Eine weitere eingegebene Zahl wird in der Variablen b gespeichert.
14
15     int summe = a + b;
16     std::cout << "Die Summe von " << a << " und " << b << " ist " << summe << "." << std::endl;
17
18     return 0;
19 }
```

Listing 6: Addition von zwei über die Eingabekonsole gegebenen Zahlen

Man kann auch, ähnlich wie bei der Ausgabe mit `std::cout`, gleich mehrere Einleseoperationen hintereinander ausführen. Im obigen Beispiel könnte man das Einlesen auch so implementieren:

```

1 std::cout << "Bitte zwei ganze Zahlen eingeben (getrennt durch Leerzeichen): ";
2 int a, b;
3 std::cin >> a >> b; //Hier werden zwei Zahlen eingelesen.
```

4.5 Konstanten

Wenn eine Variable immer nur einen bestimmten Wert haben soll, sie also als Konstante benutzt werden soll, kann man mit dem Schlüsselwort `const` vor dem Datentyp verhindern, dass ihr Wert später noch (versehentlich) geändert wird. So führt zum Beispiel das folgende Code-Fragment zu einem Compiler-Fehler:

```

1  const double pi = 3.14159;
2  pi = 2.71828; // Ein Fehler: Der Wert von pi kann nicht mehr geaender werden.

```

Es ist durchaus nützlich und empfehlenswert, `const` zu verwenden, um einen Überblick zu haben, welche Daten noch geändert werden und um ein versehentliches Überschreiben zu verhindern. Wir werden später noch andere Anwendungen von `const` sehen.

4.6 If-Else-Statement

Einfache Rechenoperatoren erlauben uns nicht, komplexe Algorithmen zu implementieren – es fehlt die Möglichkeit, abhängig vom *Ergebnis* einer Operation *unterschiedlichen* Code auszuführen. Um dies zu ermöglichen, lernen wir nun das erste Programmierstatement kennen: Das If-Else-Konstrukt:

```

if (BEDINGUNG) { ANWEISUNGSBLOCK 1 }
else { ANWEISUNGSBLOCK 2 }

```

wobei die Bedingung ein Ausdruck ist, der ein `bool` zurückliefert, und in den Anweisungsblöcken jeweils eine beliebige Folge von Befehlen stehen kann (eingerahmt durch geschweifte Klammern `{ }`). Es wird der erste Block ausgeführt, sofern die Bedingung den Wert `true` hat. Ansonsten, falls durch `else` angegeben, der zweite.

4.7 Logische und Vergleichsoperatoren

Für die Bedingung im If-Else-Statement lernen wir noch einige weitere Operatoren kennen, die sogenannten *Vergleichsoperatoren*:

Operator	Syntax
Prüfen auf Gleichheit	<code>a == b</code>
Prüfen auf Ungleichheit	<code>a != b</code>
Prüfen, ob <code>a</code> echt größer als <code>b</code> ist	<code>a > b</code>
Prüfen, ob <code>a</code> echt kleiner als <code>b</code> ist	<code>a < b</code>
Prüfen, ob <code>a</code> größer oder gleich <code>b</code> ist	<code>a >= b</code>
Prüfen, ob <code>a</code> kleiner oder gleich <code>b</code> ist	<code>a <= b</code>

Tabelle 5: Vergleichoperatoren

Diese Operatoren liefern immer die `bool`-Werte `true` oder `false` (bzw. 1 oder 0) abhängig vom Ergebnis des Vergleiches. Man beachte, dass das Prüfen auf Gleichheit mit `==` erfolgt, während `=` der Zuweisungsoperator ist. Die Verwechslung dieser Operatoren ist eine beliebte Fehlerquelle. Damit wird das If-Else-Statement bereits zu einem mächtigen Werkzeug. Ein sehr einfaches Beispiel sieht so aus:

```

1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Bitte eine ganze Zahl eingeben: ";
6      int a;
7      std::cin >> a;
8

```

```

9  if(a > 0)
10 {
11     std::cout << "Diese Zahl ist positiv." << std::endl;
12 }
13 else
14 {
15     std::cout << "Diese Zahl ist nicht positiv." << std::endl;
16 }
17
18 return 0;
19 }

```

Listing 7: Eine einfache if-Abfrage mit else-Teil

Ausdrücke vom Typ `bool` lassen sich durch logische Operatoren `and` und `or` miteinander verknüpfen und mit dem Operator `not` negieren. Die Ergebnisse der logischen Operatoren lassen sich am einfachsten durch Wertetabellen veranschaulichen. Siehe dazu Tabelle 6.

A	B	A and B	A or B	not A
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Tabelle 6: Logische Operatoren

Ein Verknüpfung von zwei Bedingungen mit `and` kann so aussehen:

```

1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Bitte eine ganze Zahl eingeben: ";
6      int a;
7      std::cin >> a;
8
9      if(a > 10 and a < 17)
10 {
11     std::cout << "Diese Zahl liegt zwischen 10 und 17." << std::endl;
12 }
13
14 return 0;
15 }

```

Listing 8: Eine if-Abfrage mit Verknüpfung von zwei Bedingungen

Verknüpft man mehr Bedingungen, kann und sollte man die gewünschte Auswertungsreihenfolge durch Klammern angeben. Zum Beispiel so (hier sollten `a` und `b` beispielsweise Variablen vom Typ `int` sein):

```

1  if((a >= 0 and b < 0) or (a < 0 and b >= 0))
2  {
3      std::cout << "Genau eine der beiden Zahlen ist negativ." << std::endl;
4  }

```

Es gibt jedoch noch eine wichtige Eigenart dieser Operatoren zu erwähnen: Die logischen Operatoren werten nur so viele ihrer Argumente aus, bis das Ergebnis der Verknüpfung bereits feststeht. So würde etwa bei der Auswertung von `(1 or x--)` die Variable `x` *nicht* dekrementiert, da das Ergebnis der Operation bereits bei der Auswertung von `1` feststeht. Dies ist selbstverständlich nur von Bedeutung, sofern eine der auszuwertenden Expressions einen Nebeneffekt hat.

4.8 Der Schleifen erster Teil: while

Wollen wir einen bestimmten Codeblock mehrfach ausführen, so verwenden wir ein Statement, was als *Schleife* bezeichnet wird. Eine Schleife wiederholt die Befehle so lange, wie ein bestimmter Ausdruck zu `true` ausgewertet wird. Die Syntax

```
while (BEDINGUNG) {ANWEISUNGSBLOCK}
```

weist den Computer an, zu prüfen, ob die Expression `BEDINGUNG` gleich `true` ist. Ist dies der Fall, so werden die Befehle im Anweisungsblock ausgeführt und wir fangen wieder von vorne mit dem Prüfen der Bedingung an. Andernfalls wird die Schleife beendet. Meistens sollten die Befehle dafür sorgen, dass `BEDINGUNG` irgendwann zu `false` ausgewertet, indem etwa Variablen verändert werden.

Wir wollen ein Beispiel angeben, welches die Geometrische Reihe $\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}$ ausrechnet:

```
1 double q = 0.2;
2 double x = 1.0, y = 0.0; /* Hilfsvariablen */
3 while (x > 1e-10)        /* Solange x nicht zu klein ist */
4 {
5     y = y + x;            /* y speichert die Partialsummen */
6     x = x * q;            /* Berechne den n"achsten Summanden */
7 }
8 /* Ergebnis steht jetzt in y */
```

Listing 9: Geometrische Reihe

Dieses Beispiel zeigt anschaulich, dass Programme deutlich aufwendiger sein können, als sie müssen. Wir hätten ebenso gut `y = 1.0/(1.0 - q);` schreiben können, was der Computer in einem Bruchteil der Zeit berechnen könnte. Man sollte sich immer bemühen, nicht unnötig Rechenzeit zu vergeuden.

Wenn man die Befehle im `ANWEISUNGSBLOCK` gerne ausführen möchte, bevor das erste Mal geprüft wird, ob `BEDINGUNG` zu `false` ausgewertet, so kann man eine `do-while`-Schleife verwenden:

```
do {ANWEISUNGSBLOCK} while(BEDINGUNG);
```

Man bemerke hier das zwingend erforderliche Semikolon am Ende.

4.9 Der Schleifen zweiter Teil: for

Die `while`-Schleife lässt sich verallgemeinern zur `for`-Schleife, dem folgenden Konstrukt:

```
for( INITIALISIERUNG; BEDINGUNG; STEP )
{ ANWEISUNGSBLOCK }
```

wobei wir dies wie folgt durch eine `while`-Schleife modellieren könnten, sofern die Bedingung angegeben ist:

```
INITIALISIERUNG;
while ( BEDINGUNG )
{
    ANWEISUNGSBLOCK
    STEP;
}
```

Damit sind also die Initialisierung, der Step und die Bedingung jeweils eine Expression. Beispielsweise gibt die folgende Schleife die ganzen Zahlen von -5 bis 9 aus:

```
1 for(int i = -5; i < 10 ; ++i)
2 {
3     std::cout << i << std::endl;
4 }
```

Das Beispiel aus dem letzten Abschnitt kann man so umschreiben:

```
1 double y = 0.0;
2 for (double x = 1.0, q = 0.2; x > 1e-10; x = x * q)
3 {
4     y = y + x;
5 }
6 std::cout << y << std::endl;
```

Listing 10: Geometrische Reihe mit einer for-Schleife

Man beachte, dass die Variablen `x` und `q` hier in der Initialisierung der `for`-Schleife deklariert und initialisiert werden. Die Variable `y` wird dagegen außerhalb der Schleife definiert. Letzteres ist notwendig, um auf die Variable auch außerhalb der Schleife (hier bei der Ausgabe) zugreifen zu können. Die Variablen `x` und `q`, die in der Schleife definiert werden, können außerhalb nicht benutzt werden. So würde es zu einem Compiler-Fehler führen, wenn man die letzte Zeile in obigem Fragment durch `std::cout << q << std::endl;` oder `std::cout << x << std::endl;` ersetzte.

Generell gilt: Wenn eine Variable innerhalb eines Blocks definiert wird, dann ist sie auch nur innerhalb des Blocks verfügbar. Also würde folgender Code zu einem Fehler führen:

```
1 #include<iostream>
2
3 int main()
4 {
5     int x;
6
7     std::cin >> x;
8
9     if(x == 37)
10    {
11        int y = x + 5;
12        std::cout << y << std::endl; // Korrekt: Hier kann man auf y zugreifen
13    }
14    std::cout << y << std::endl; // Fehler: Hier kann man nicht auf y zugreifen
15 }
```

Ebenso falsch ist:

```
1 for(int i = -5; i < 10 ; ++i)
2 {
3     std::cout << i << std::endl;
4 }
5 std::cout << i << std::endl; // Fehler: Hier kann man nicht auf i zugreifen.
```

Der Bereich, in dem eine Variable sichtbar ist heißt ihr *Sichtbarkeitsbereich* oder *Scope*.

Lässt man bei der `for`-Schleife die Bedingung weg, bricht die Schleife nicht ab. Genauer: Die Schleife verhält sich so, als wäre die Bedingung die konstante Expression `true`. Step oder Initialisierung sind ebenfalls optional und können weggelassen werden – also ist folgende Schleife eine Endlosschleife: `for(;;);`

Es gibt zwei besondere Statements, welche innerhalb von Schleifen verwendet werden können:

Statement	Effekt
<code>break;</code>	Schleife abbrechen bzw. zum nächsten Statement nach der Schleife springen. In einer <code>for</code> -Schleife wird der Step nicht noch einmal ausgeführt.
<code>continue;</code>	Nur diesen Schleifendurchlauf abbrechen (zum Step springen).

Tabelle 7: Spezielle Schleifenbefehle

Bei einer `for`-Schleife sorgt ein `continue`-Statement also dafür, dass der Step noch ausgeführt wird, bevor die Bedingung abgefragt wird und dann evtl. der nächste Schleifendurchlauf beginnt.

4.10 Eigene Bezeichnungen für Datentypen

Wenn man in einer Variable statt einer Zahl einen Zustand abspeichern will (z.B. die Farbe eines Gegenstandes oder den heutigen Wochentag), so kann man das erreichen, indem man sich statt dessen z.B. einen `int` speichert und sich merkt, dass beispielsweise 0 für Montag steht, 1 für Dienstag usw. In C++ gibt es aber mit `enum` ein Schlüsselwort, das einem diese Zuordnung erleichtert.

```
enum NAME {ZUSTAND_0, ZUSTAND_1, ..., ZUSTAND_k};
```

```
1 #include<iostream>
2
3 enum Farbe {rot , orange , gelb , gruen , blau}; // Hier wird der Typ Farbe definiert.
4
5 int main()
6 {
7     Farbe meine_farbe = blau;
8     std::cout << meine_farbe << std::endl;
9
10    meine_farbe = rot;
11    std::cout << meine_farbe << std::endl;
12
13    return 0;
14 }
```

Listing 11: Ein Beispiel für die Benutzung von `enum`

Gespeichert werden die Variablen von den so erzeugten Typen als `int`. Daher ist auch eine direkte Umwandlung, von einem `enum`-Typ in ein `int` möglich. Genau dies geschieht in obigem Beispiel bei der Ausgabe mit `std::cout`. Die beiden Farb-Werte werden als `int` interpretiert, weshalb die Ausgabe des Programms in den Zahlen 4 und 0 besteht.

Man beachte aber, dass eine Umwandlung eines `int` in einen `enum`-Typ nicht möglich ist. In obigem Beispiel würde deshalb

```
1 int i = 2;
2 Farbe farbe = i; // Fehler: Diese Zuweisung geht nicht.
```

zu einem Fehler führen.

Man kann `enums` gut dafür benutzen, um auf einzelne Einträge in einem Array zuzugreifen, die z.B. eine geometrische Bedeutung haben:

```
1 #include<iostream>
2
3 enum Dimension {x_dim, y_dim, z_dim}; // Hier wird der Typ Dimension definiert.
4
5 int main()
6 {
```

```

7   double point[3] = {4, 7, 23};
8
9   std::cout << "x_coor: " << point[x_dim] << std::endl;
10  std::cout << "y_coor: " << point[y_dim] << std::endl;
11  std::cout << "z_coor: " << point[z_dim] << std::endl;
12
13  return 0;
14 }

```

Listing 12: Ein weiteres Beispiel für die Benutzung von enum

Mit dem Schlüsselwort `using` kann man Datentypen eigene Namen geben, beispielsweise so:

```

1  #include <iostream>
2
3  using myint = unsigned long; // Hier wird myint als andere Bezeichnung fuer unsigned long definiert.
4
5  int main()
6  {
7      myint x = 37;
8      myint y = 42;
9
10     std::cout << "x: " << x << " y: " << y << std::endl;
11
12     return 0;
13 }

```

Listing 13: Der Datentyp `unsigned long` erhält einen zweiten Namen

In dem obigen Beispiel kann `myint` äquivalent zu `unsigned long` verwendet werden.

Es stellt sich die Frage, warum man einem Datentypen einen weiteren Namen geben soll. Zum einen erlaubt diese Schreibweise, dass man, wenn man später in einem Programmteil zum Beispiel den Typ `int` durch den Typ `double` ersetzen will, nur an einer Stelle etwas ändern muss. Außerdem kann man den Typen sprechende Namen geben, die Auskunft darüber geben, wozu Variablen von diesem Typ verwendet werden sollen (also zum Beispiel `using coor = int`, wenn man ganzzahlige Koordinaten verwalten will).

4.11 Laufzeitmessung

Eine einfache Art, die Laufzeit eines Programmabschnitt zu messen, ist in dem folgenden Beispiel dargestellt:

```

1  #include <iostream>
2  #include <ctime>
3
4  int main()
5  {
6      clock_t startzeit, endzeit;
7      long differenz;
8
9      startzeit = clock(); // Zeitmessung 1
10
11     long a = 6;
12
13     for(int i = 0; i < 100000000; ++i)
14     {
15         a += (i);
16     }
17
18     endzeit = clock(); // Zeitmessung 2
19
20     differenz = endzeit - startzeit; // Zeitdifferenz zwischen den beiden Messungen
21
22     std::cout << "Differenz: " << differenz << std::endl;

```

```

23  std::cout << "Zeit in Sekunden: " << static_cast<double>(differenz)/CLOCKS_PER_SEC << std::endl;
24
25  std::cout << a << std::endl;
26 }

```

Listing 14: Laufzeitmessung.

Mit der Funktion `clock()` (deren Ausgabe vom Typ `clock_t` ist) lässt sich die aktuelle Zeit abfragen. Mit zwei solche Messungen lässt sich also bestimmen, wie viel Zeit in einem bestimmten Code-Abschnitt verbraucht wurde. Diese Differenz lässt sich mit der Konstante `CLOCKS_PER_SEC` in Sekunden umwandeln. Diese Methode stammt noch aus der C-Welt, daher werden die Bestandteile mit `#include <ctime>` eingebunden.

4.12 Zufallszahlen

Um Pseudo-Zufallszahlen (die natürlich in Wirklichkeit vollständig deterministisch sind) zu erzeugen, gibt es auch ein einfaches Art, die aus der C-Standardbibliothek stammt, die im folgenden Beispiel `#include <cstdlib>` eingebunden wird:

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4
5  int main()
6  {
7      srand(time(NULL));
8
9      for(int i=0; i < 10; i++)
10     {
11         std::cout << rand()%100 << std::endl;;
12     }
13
14     return 0;
15 }

```

Listing 15: Erzeugung von Zufallszahlen.

Hier werden mit der Funktion `rand()` Zufallszahlen von 0 bis `RAND_MAX` erzeugt, wobei `RAND_MAX` eine Konstante ist, die in `cstdlib` spezifiziert ist. Der Rückgabewert von `rand()` ist ein `int`. Die Folge der von `rand()` erzeugten Zufallszahlen wäre immer gleich, aber der Aufruf `srand(n)`; (für einen `int`-Wert `n`) sorgt dafür, dass man für jeden Wert von `n` eine andere Folge bekommt. Wenn man, wie im Beispiel, der Funktion `srand()` die aktuelle Zeit mit `time(NULL)` übergibt, erhält man in jedem Lauf eine andere Folge von Zufallszahlen. Mit `rand()%100` kann man außerdem erzwingen, dass die Werte der Zufallszahlen nicht größer als 100 sind.

5 Funktionen

Funktionen sind ein grundlegendes und wichtiges Konzept der Programmierung. Sie ermöglichen es, häufig benötigte Programmzeilen als “Unterprogramm” zusammenzufassen. An anderen Stellen im gleichen Programm kann man dann durch einen sogenannten *Aufruf* der Funktion dorthin verzweigen. In der Funktion selbst kann man durch das **return** - Statement dafür sorgen, dass die Ausführung an der Stelle fortgesetzt wird, an der die Funktion aufgerufen wurde. Wie ihre mathematischen Äquivalente können Funktionen Argumente erhalten und einen Rückgabewert besitzen.

5.1 Funktionsdefinitionen

Eine *Funktionsdefinition* hat folgende Form:

```
RÜCKGABETYP FUNKTIONSNAME(  
    PARAMETERTYP 1 PARAMETERNAME 1,  
    PARAMETERTYP 2 PARAMETERNAME 2,  
    ...,  
    PARAMETERTYP n PARAMETERNAME n ) {  
    BEFEHLE  
}
```

Der Rückgabetyt ist hierbei ein beliebiger Datentyp - dieser bestimmt, welchen Datentyp der Ausdruck des Funktionsaufrufes hat. Ein *Funktionsaufruf* hat die Syntax:

```
FUNKTIONSNAME ( PARAMETER 1, ..., PARAMETER n )
```

Dies bedeutet, dass eine Funktion ein vom Programmierer neu definierter Operator ist: Sie weist einem oder mehreren Werten einen neuen Wert (den Rückgabewert) zu.

Bei jedem Funktionsaufruf werden zunächst neue Variablen PARAMETERNAME 1 bis PARAMETERNAME n erstellt, welche vom in der Funktionsdefinition angegebenen Datentyp sind. Dann werden die Expressions PARAMETER 1 bis PARAMETER n ausgewertet und den Variablen in der entsprechenden Reihenfolge zugewiesen. Anschließend werden die Befehle in der Funktionsdefinition ausgeführt, bis der Wert berechnet wurde, den der Funktionsaufruf haben soll. Durch das folgende Statement beendet die Funktion sich selbst augenblicklich und legt ihren sogenannten *Rückgabewert* fest: Der Wert des Funktionsaufrufes.

```
return RÜCKGABEWERT;
```

Die Parameter in der Funktionsdefinition sind Variablendeklarationen, deren Initialisierung durch den Funktionsaufruf stattfindet. Sie gehören zum Block der Funktionsdefinition und können (sollten) dort zur Berechnung des Rückgabewerts verwendet werden. Dennoch kann eine Funktion selbstverständlich zu Beginn weitere, interne Variablen erstellen.

Innerhalb der Funktion sind dies aber insgesamt die *einzigsten* Variablen, auf die direkt (mit Namen) zugegriffen werden kann. Wir wollen nun Code für eine Funktionsdefinition vorstellen, welche das Signum einer Ganzzahl ausrechnet (siehe auch 4.7) und diese Funktion dann aufrufen:

```

1 #include <iostream>
2
3 int sign(int x)
4 {
5     if (x < 0)
6     {
7         return -1;
8     }
9     else
10    {
11        return (x != 0);
12    }
13 }
14
15 int main()
16 {
17     std::cout << sign(-5) << std::endl; // Wird -1 ausgegeben.
18     return 0;
19 }

```

Listing 16: Beispiel für eine Funktion

Die in Listing 16 angegebene Funktion `sign(int x)` gibt in Abhängigkeit vom Vorzeichen von `x` entweder `-1`, `0` oder `1` aus. Man beachte, dass der Ausdruck `(x != 0)` vom Typ `bool` ist und dieser Ausdruck dann in ein `int` umgewandelt wird.

Noch ein Beispiel:

```

1 #include <iostream>
2
3 /* Berechnet basis hoch exponent */
4 double potenz(double basis, unsigned int exponent)
5 {
6     double ergebnis = 1.0;
7
8     while(exponent > 0)
9     {
10        ergebnis *= basis;
11        exponent--;
12    }
13    return ergebnis;
14 }
15
16 int main()
17 {
18     std::cout << potenz(0.5, 4) << std::endl; // Wird 0.0625 ausgegeben.
19     return 0;
20 }

```

Listing 17: Funktion zum Berechnen ganzer Potenzen von Fließkommazahlen

Wir lernen an dieser Stelle noch einen neuen Datentyp kennen, den Datentyp `void`. Man kann keine `void`-Variablen deklarieren, denn eine Expression mit Datentyp `void` hat *keinen* Wert. Allerdings gibt es Funktionen mit *Rückgabotyp* `void`, welche man auch als Prozeduren bezeichnet. Eine Prozedur muss kein `return`-Statement enthalten, kann jedoch das leere `return`-Statement `return;` verwenden, um sich selbst zu beenden.

```

1 #include <iostream>
2
3 void gib_zahl_aus(int x)
4 {
5     std::cout << "Die Zahl lautet: " << x << std::endl;
6     return; // Das muesste hier nicht stehen.
7 }
8
9 int main()
10 {
11     gib_zahl_aus(37);

```

```

12     return 0;
13 }

```

Listing 18: Beispiel für eine (mäßig nützliche) Prozedur

Funktionen lassen sich nicht nur von der `main`-Funktion aus aufrufen, sondern sie können auch von anderen Funktionen oder auch von sich selbst aufgerufen werden. Schauen wir uns dafür ein Beispiel an. Die Folge $(f_n)_{n \in \mathbb{N}}$ der Fibonacci-Zahlen ist wie folgt rekursiv definiert: Es gelten $f_0 = 1$ und $f_1 = 1$, und für $n \geq 2$ gilt $f_n = f_{n-1} + f_{n-2}$. Wenn wir die einzelnen Folgenglieder berechnen wollen, ist es naheliegend, einen rekursiven Ansatz zu wählen, wie in dem folgenden Programmbeispiel:

```

1  #include <iostream>
2
3  unsigned fibonacci(unsigned n)
4  {
5      if (n < 2)
6      {
7          return 1;
8      }
9      else
10     {
11         return fibonacci(n-1) + fibonacci(n-2);
12     }
13 }
14
15 int main()
16 {
17     std::cout << "Bitte eine nichtnegative Zahl: " << std::endl;
18     unsigned n;
19     std::cin >> n;
20
21     std::cout << "f(" << n << ") ist " << fibonacci(n) << std::endl;
22
23     return 0;
24 }

```

Listing 19: Die Fibonacci-Zahlen, über eine rekursive Funktion berechnet.

Die Funktion `fibonacci` ruft sich in diesem Beispiel selbst mit kleineren Werten von `n` auf. Man beachte allerdings, dass diese Art, die Fibonacci-Zahlen zu berechnen, furchtbar ineffizient ist.

5.2 Variablenübergabe per Referenz

Man beachte, dass bei der Übergabe einer Variable an eine Funktion eine Kopie dieser Variable erzeugt wird. Der Wert dieser Kopie stimmt natürlich mit dem Wert der gegebenen Variable überein, die kopierte Variable steht aber an einer anderen Speicherstelle. Insbesondere wirken sich Änderungen am Wert der Kopie nicht auf den Wert der Originalvariable aus. Hier ist ein Beispiel:

```

1  #include <iostream>
2
3  int addiere_37(int x) // Eine Funktion, die zu einer gegebenen ganzen Zahl 37 addiert.
4  {
5      x = x + 37;
6      std::cout << "Wert von x ist " << x << std::endl;
7      return x;
8  }
9
10 int main()
11 {
12     int a = 42;
13     int result;
14

```

```

15     result = addiere_37(a);
16     std::cout << "Wert von a ist " << a << std::endl;
17     std::cout << "Wert von result ist " << result << std::endl;
18     return 0;
19 }

```

Listing 20: Beispiel für eine Funktion, die eine “by value” gegebene Variable ändert.

Die Ausgabe dieses Programms lautet wie folgt:

Wert von x ist 79

Wert von a ist 42

Wert von result ist 79

Die in der Funktion benutzte Variable **x** ist eine Kopie der Variable **a**. Dass der Wert von **x** innerhalb der Funktion verändert wird, hat also keine Auswirkungen auf den Wert von **a** in der Hauptfunktion.

Man kann verhindern, dass eine Variable beim Aufruf einer Funktion kopiert wird, indem man sie per Referenz übergibt. Die Syntax ist folgende:

PARAMETERTYP &PARAMETERNAME

Vor dem Namen der Parameters steht also in der Funktionsdeklaration ein **&**-Zeichen. Am Funktionsaufruf ändert sich nichts. Wenn man eine Referenz auf eine Variable übergibt, wird deren Inhalt im Speicher nicht kopiert, und die Variable innerhalb der Funktion bezieht sich auf denselben Speicherbereich.

In obigem Beispiel kann man also die Schnittstelle der Funktion wie folgt ändern:

```

1 int addiere_37(int &x) // Eine Funktion, die zu einer gegebenen ganzen Zahl 37 addiert.

```

Die Ausgabe sieht dann so aus:

Wert von x ist 79

Wert von a ist 79

Wert von result ist 79

Auf diesem Wege kann man nun z.B. die Werte von zwei Variablen vertauschen:

```

1 #include <iostream>
2
3 /* Vertauscht die Werte der Variablen a und b */
4 void swap(int &a, int &b)
5 {
6     int temp = a;
7     a = b;
8     b = temp;
9 }
10
11 int main()
12 {
13     int a = 10, b = 7;
14
15     swap(a, b);
16
17     std::cout << "a: " << a << std::endl;
18     std::cout << "b: " << b << std::endl;
19
20     return 0;
21 }

```

Listing 21: Ein Funktion, welche die Werte von zwei Variablen vertauscht.

Wenn man eine Variable wie in `int addiere_37(int &x)` als Referenz übergibt, kann der Wert der Variable innerhalb der Funktion (in diesem Fall `addiere_37`) geändert werden. Unter Umständen will man dies verhindern. In dem Fall hilft wieder das Schlüsselwort `const` weiter, das wir schon in Abschnitt 4.5 gesehen haben. Man kann z.B. in obigem Programm folgendes setzen:

```
1 int addiere_37(const int &x) // Eine Funktion, die zu einer gegebenen ganzen Zahl 37 addiert.
```

Dies führt dazu, dass man bei einem Aufruf von `addiere_37` eine Referenz auf die Variable übergibt (diese wird also nicht im Speicher kopiert), aber in der Funktion keine Änderungen am Wert vorgenommen werden können. Der folgende Code würde deshalb zu einem Compiler-Fehler führen.

```
1 int addiere_37(const int &x) // Eine Funktion, die zu einer gegebenen ganzen Zahl 37 addiert.
2 {
3     x = x + 37; //Fehler: x darf nicht geändert werden.
4     std::cout << "Wert von x ist " << x << std::endl;
5     return x;
6 }
```

Möglich wäre aber:

```
1 int addiere_37(const int &x) // Eine Funktion, die zu einer gegebenen ganzen Zahl 37 addiert.
2 {
3     std::cout << "Wert von x ist " << x << std::endl;
4     return x + 37;
5 }
```

Warum sollte man nun überhaupt eine Variable per Referenz übergeben, wenn man mit dem Schlüsselwort `const` trotzdem verhindern will, dass sie geändert wird? Dafür kann es zwei Gründe geben:

- Erstens wird verhindert, dass man im Inneren der Funktion die Variable versehentlich mit der Absicht ändert, einen Effekt auf die übergebene Variable zu haben.
- Durch die Übergabe per Referenz wird verhindert, dass die Variable kopiert werden muss. Bei Datentypen wie `int` ist ein solches Kopieren kein Problem, weil es schnell geht und kaum Speicherplatz benötigt wird. Wir werden aber auch sehen, wie sich sehr komplexe Datentypen erzeugen lassen. Wenn diese bei jedem Funktionsaufruf kopiert werden, kostet das Speicherplatz und insbesondere Laufzeit. Darauf werden wir zurückkommen, wenn wir Klassen behandeln.

5.3 Funktionsdeklaration vs. Funktionsdefinition

Möchte man eine Funktion aufrufen, so muss die Definition dieser Funktion im Quellcode vor dem Funktionsaufruf liegen, da der Compiler die aufzurufende Funktion bereits “kennen” muss, damit er einen Aufruf korrekt in Maschinencode übersetzen kann: Dazu muss er wenigstens wissen, wie genau die Funktionsargumente und der Rückgabetyt aussehen. Man kann diese Informationen jedoch angeben, bevor man die Funktion tatsächlich definiert, indem man lediglich eine *Funktionsdeklaration* verwendet. Dieses Statement sieht wie folgt aus:

```
RÜCKGABETYP FUNKTIONSNAME(
    PARAMETERTYP 1 PARAMETERNAME 1,
```

```

    ...,
    PARAMETERTYP n PARAMETERNAME n );

```

Die Deklaration enthält also nur den sogenannten *Funktionskopf*, in dem alle für den Compiler wichtigen Informationen enthalten sind. Nachdem die Funktion deklariert ist, kann man sie im nachfolgenden Quellcode verwenden. An irgendeiner Stelle muss allerdings dann die tatsächliche Definition stehen. Hier ein Beispiel, welches ohne dieses Sprachkonstrukt gar nicht möglich wäre:

```

1 #include <iostream>
2
3 // Achtung: Die folgende Implementierung ist komplett hanebuechen!
4
5 // Funktionsdeklarationen
6 int ungerade(int); // diese Deklaration ist notwendig.
7 int gerade(int); // diese Deklaration nicht, ist aber huebsch.
8
9 /* Funktionsdefinitionen */
10 int gerade(int n) // testet, ob n gerade ist
11 {
12     if (n == 0)
13     {
14         return 1;
15     }
16     else
17     {
18         return ungerade(n-1); /* wir m"ussen "ungerade" kennen */
19     }
20 }
21 int ungerade(int n) { /* testet, ob n ungerade ist */
22     if (n == 0)
23     {
24         return 0;
25     }
26     else
27     {
28         return gerade(n-1);
29     }
30 }
31
32 int main()
33 {
34     int n;
35     std::cout << "Eine nichtnegative ganze Zahl bitte: " << std::endl;
36     std::cin >> n;
37     if ( gerade(n) )
38     {
39         std::cout << "Diese Zahl ist gerade." << std::endl;
40     }
41     else
42     {
43         std::cout << "Diese Zahl ist ungerade." << std::endl;
44     }
45     return 0;
46 }

```

Listing 22: Funktionsdeklarationen sind notwendig

Die Umsetzung dieser Funktionen ist natürlich haarsträubend ineffizient, umständlich und unverständlich. Wir konnten jedoch kein besseres Beispiel für Funktionen finden, die sich auf diese Art und Weise gegenseitig aufrufen: Man bezeichnet dies auch als *indirekte Rekursion*.

5.4 Modulares Programmieren und Linken

Die Kompilierung von großen Programmen zu schnellem und effizientem Maschinencode bedarf eines deutlich merkbaren Rechenaufwands. Während der Weiterentwicklung oder Fehleranalyse

solcher Programme müssen allerdings ständig Teile des Programmcodes verändert werden und es wäre zu zeitaufwendig, das gesamte Programm ständig neu zu kompilieren - insbesondere, da sich ja nur gewisse Teilbereiche des Programms ändern - etwa nur eine bestimmte Funktion. Man geht deswegen dazu über, einzelne Teile eines Programms so voneinander zu trennen, dass der Compiler sie unabhängig voneinander in Maschinencode übersetzen kann. Diese Teile nennt man auch *Module*.

Nachdem ein solches Modul kompiliert wurde, ist es natürlich kein lauffähiges Programm – insbesondere verwendet das Modul unter Umständen Funktionen, deren Programmcode sich in anderen Modulen befindet. Um diese Abhängigkeiten aufzulösen, wird in der Schlussphase der Codegenerierung ein Programm (der *Linker*) gestartet, um die kompilierten Module zu einem lauffähigen Programm zusammenzufügen. Diesen Vorgang bezeichnet man dementsprechend als *Linken*. Ein Modul in C++ ist zunächst eine Datei mit Dateiendung “cpp”. Jede solche .cpp-Datei wird von dem Compiler zu einer sogenannten *Objektdatei* kompiliert, welche das kompilierte Modul darstellt. Diese Objektdatei enthält Informationen darüber, welche Funktionen das Modul enthält und welche Funktionen von dem Modul aus anderen Modulen benötigt werden. Sind einmal alle Objektdateien erstellt, löst der Linker die Abhängigkeiten zwischen ihnen auf und fügt die Objektdateien zu einem lauffähigen Programmcode zusammen. Dieser Vorgang ist unabhängig von der Kompilierung.

Bei der Kompilierung ist es jedoch erforderlich, dass Funktionen definiert werden, bevor sie im Quellcode danach verwendet werden. Existiert etwa eine Quellcodedatei `moremath.cpp`, welche unter anderem eine Funktion

```
1 unsigned fibonacci(unsigned n)
```

beinhaltet, so könnte man die folgende `main.cpp` natürlich trotzdem nicht erfolgreich kompilieren, da zumindest eine Deklaration der Funktion fehlt:

```
1 #include <iostream>
2 /* Hier fehlt eine Deklaration oder Ähnliches */
3 int main()
4 {
5     unsigned j;
6     for (j = 1; j < 10; j++)
7     {
8         std::cout << fibonacci(j) << std::endl;
9     }
10    return 0;
11 }
```

Listing 23: Fehlende Deklaration

Man mache sich klar, dass dies ein Problem des Compilers und völlig unabhängig vom Linker ist. Um dieses Problem zu lösen, gehört zu jedem Modul auch eine *Headerdatei* mit der Dateiendung “h”, welche den gleichen Namen wie die Quellcodedatei des Moduls erhält. Diese enthält nur Funktionsdeklarationen. Im Sinne des obigen Beispiels sähe die Headerdatei `moremath.h` etwa so aus:

```
1 unsigned factorial(unsigned n); /* berechnet n! */
2 unsigned fibonacci(unsigned n); /* berechnet die n-te Fibonaccizahl */
```

Listing 24: Header-Datei für das `moremath`-Modul

Also enthält die Headerdatei lediglich Informationen über die Verwendung der Funktionen, die sich im zugehörigen Modul befinden, damit eine Kompilierung mit voneinander getrenntem Code

überhaupt erst möglich wird. Mit dieser Datei ist `main.cpp` in folgender Variante nun kompilierbar:

```
1 #include <iostream>
2 # include "moremath.h"
3 int main()
4 {
5     unsigned j;
6     for (j = 1; j < 10; j++)
7     {
8         std::cout << fibonacci(j) << std::endl;
9     }
10    return 0;
11 }
```

Listing 25: Deklaration fehlt nun nicht mehr

Die Headerdateien von selbstgeschriebenen Modulen werden durch die `#include`-Anweisung direkt in den Quellcode eingefügt (kopiert). Die Headerdateien eigener Module werden mit Anführungszeichen angegeben, Headerdateien von Systemmodulen mit spitzen Klammern. In der Tat gibt es bereits im System vorhandene Module wie etwa `iostream`, welche sich in ihrer Funktionsweise nicht von selbst erstellten Modulen unterscheiden. Das Modul `moremath.cpp` könnte nun wie folgt aussehen:

```
1 #include "moremath.h"
2
3 unsigned factorial(unsigned n)
4 {
5     unsigned f;
6
7     for (f = 1; n; n--)
8     {
9         f *= n;
10    }
11    return f;
12 }
13
14 unsigned fibonacci(unsigned n)
15 {
16     if (n < 2)
17     {
18         return 1;
19     }
20     else
21     {
22         return fibonacci(n-1) + fibonacci(n-2);
23     }
24 }
```

Listing 26: Das `moremath`-Modul

Die Quellcodedatei bindet für gewöhnlich ihre zugehörige Headerdatei ein. Dies hat viele Vorteile, die in Zukunft noch klarer werden, doch einen Grund kennen wir bereits: Sollten die Funktionen eines Moduls sich gegenseitig verwenden, so vermeiden wir durch Einfügen aller Deklarationen zu Anfang Compilerfehler.

Um mehrere `cpp`-Dateien zu kompilieren kann man sie einfach alle in den Compiler-Aufruf aufnehmen, also z.B.:

```
g++ -std=c++11 -Wall -Wpedantic main.cpp more_math.cpp
```

Zusammenfassung: Der Compiler ist während der Kompilierung lediglich auf vollständige Deklarationen aller verwendeten Funktionen angewiesen. Diese befinden sich in den jeweiligen Headerdateien. Ist die Kompilierung abgeschlossen, muss der Linker aus einer Menge von kompilierten

Modulen ein Programm erstellen. Dazu sucht er zunächst das Modul, welches die `main`-Funktion enthält, da an dieser Stelle die Ausführung des Programms beginnen soll. Von diesem Modul ausgehend sucht der Linker nun zu jedem noch nicht verknüpften Funktionsnamen in allen Modulen (auch den Systemmodulen) nach einer Funktion mit dem gleichen Namen und bindet jenes Modul ein, sobald er es gefunden hat. Dies wird fortgeführt, bis alle Namen aufgelöst sind und ein lauffähiges Programm erstellt werden kann.

Es sei an dieser Stelle noch einmal betont, dass das Konzept von Headerdateien (`.h`) ein Modul auf Compilerebene beschreibt, während die Aufteilung von Funktionen auf verschiedene Quellcodedateien (`.cpp`) ein Modul auf Linkerebene beschreibt. Diese beiden Konzepte funktionieren unabhängig voneinander. Eine Headerdatei könnte etwa Deklarationen von Funktionen enthalten, die auf zwei Quellcodedateien verteilt sind, oder man könnte Deklarationen von Funktionen einer Quellcodedatei auf mehrere Headerdateien verteilen. Auch die Namen von Header- und Quellcodedatei eines Moduls *müssen* streng genommen nicht übereinstimmen - all dies gebietet nur der gute Stil und die Übersichtlichkeit des gesamten Projekts.

5.5 Der Präprozessor

Bevor der Compiler tatsächlich mit der Kompilierung eines C++-Programms beginnt, wird ein Programm aufgerufen, das als Präprozessor bezeichnet wird. Er führt ausschließlich Textersetzungen im Quellcode durch. Er kann durch spezielle Befehle im Quellcode gesteuert werden, welche durch eine führende Raute (`#`) gekennzeichnet werden. Einige dieser Befehle kennen wir bereits, etwa geschieht das Einbinden von Headerdateien durch den Präprozessorbefehl:

```
1 #include <stdlib.h>
2 #include "myheader.h"
```

Listing 27: Einbinden von Header-Dateien sind Präprozessoranweisungen

Hier erfolgt eine reine Textersetzung – der Inhalt der Datei `myheader.h` wird vollständig an die Stelle des `include`-Befehls kopiert. Die spitzen Klammern sind notwendig, um eine Standardheader einzufügen, während Anführungszeichen verwendet werden, um selbst erstellte Headerdateien einzufügen. Es gibt jedoch noch einige weitere nützliche Präprozessorbefehle.

5.5.1 Makrodefinition

```
#define MAKRO REPLACE
```

ist eine sogenannte Makrodefinition. Sie weist den Präprozessor an, die Zeichenkette `MAKRO` im Folgenden immer durch `REPLACE` zu ersetzen. Dabei kann `REPLACE` auch der leere String sein bzw. weggelassen werden. Dies kann etwa dazu genutzt werden, Konstanten zu definieren:

```
1 #define PI 3.1415926535897932
```

Empfehlenswert ist diese Art der Konstantendefinition aber im Allgemeinen nicht, man sollte lieber mit `const` angelegt Variablen benutzen.

5.5.2 Bedingte Texte

```
#if AUSDRUCK
    TEXT A
#endif
```

```
#else
    TEXT B
#endif
```

Dieser Befehl erlaubt es uns, mit dem Präprozessor kleinere Fallunterscheidungen durchzuführen. Wenn die Bedingung der `if` - Anweisung erfüllt ist, so wird Text A eingefügt, andernfalls Text B. Der `else` - Zweig der Anweisung ist optional. Auf die verschiedenen Möglichkeiten für Ausdrücke lohnt es sich kaum, hier einzugehen - der wichtigste Ausdruck ist vermutlich

```
#if defined(MAKRONAME)
```

welcher prüft, ob ein Makro mit Namen `MAKRONAME` bereits definiert ist. Damit lassen sich insbesondere Inklusionskreise bei Headerdateien vermeiden:

```
1 #if !defined(MYMATH_H)
2 #define MYMATH_H
3 /* Inhalt */
4 #endif
```

Listing 28: Zirkuläre Inklusion verhindern

Beim ersten Einfügen dieser Datei mittels `#include` wird das Makro `MYMATH_H` noch unbekannt sein, daher wird der Präprozessor den Text nach `#if` einfügen und insbesondere das Makro `MYMATH_H` definieren. Sollte die Datei ein zweites mal per `#include` eingefügt werden, ist das Makro `MYMATH_H` nun definiert und der Präprozessor überspringt alles zwischen `#if` und `#endif`. Damit ist also sichergestellt, dass der Inhalt einer Headerdatei nur ein einziges Mal in einem Projekt eingefügt wird. Man nennt dieses Konstrukt auch *Include Guards* (Include-Wächter). Es sollte nach Möglichkeit bei allen Headerdateien verwendet werden, da der Präprozessor sonst in eine Endlosschleife gerät, sobald zwei Headerdateien sich gegenseitig per `#include` einbinden. Da dieser Befehl überaus nützlich und weit verbreitet ist, gibt es eine Kurzschreibweise:

```
#ifndef MYMATH_H ⇒ #if !defined(MYMATH_H)
#define MYMATH_H ⇒ #if defined(MYMATH_H)
```

5.6 Namensräume

Funktionen sollten immer sprechende Namen haben, die es erlauben, schnell zu erkennen, was sie tun. Man kann dazu auch Gruppen von Funktionen, die zusammengehören mit einem einheitlichen Präfix versehen. Dazu definiert man einen sogenannten Namensraum bzw. `namespace`. Im folgenden Beispiel wird ein Namensraum mit der Bezeichnung `Hugo` definiert:

```
1 # include<iostream>
2
3 void funktion_1()
4 {
5     std::cout << "Dies ist Funktion 1" << std::endl;
6 }
7
8 namespace Hugo
9 {
10     void funktion_1()
11     {
12         std::cout << "Dies ist die Hugo-Funktion 1" << std::endl;
13     }
14
15     void funktion_2()
```

```

16 {
17     funktion_1();
18     std::cout << "Dies ist die Hugo-Funktion 2" << std::endl;
19 }
20 }
21
22 int main()
23 {
24     Hugo::funktion_1();
25     Hugo::funktion_2();
26     funktion_1();
27 }

```

Listing 29: Ein Beispiel für einen namespace.

Die darin definierten Funktionen `funktion_1()` und `funktion_2()` können nun von außen durch `Hugo::funktion_1()` bzw. `Hugo::funktion_2()` aufgerufen werden. Innerhalb des Namensraumes kann man diese Präfixe weglassen. Selbst wenn es außerhalb des Namensraums Funktionen mit demselben Namen gibt (wie hier z.B. `funktion_1()`), wird innerhalb des Namensraums die Version aus dem Namensraum selbst genommen.

Die Ausgabe lautet deshalb in diesem Beispiel so:

```

Dies ist die Hugo-Funktion 1
Dies ist die Hugo-Funktion 1
Dies ist die Hugo-Funktion 2
Dies ist Funktion 1

```

Eine Anwendung von Namensräumen haben wir schon in der Standard-Bibliothek gesehen, bei der alle zugehörigen Funktionen (und Klassen) mit `std::` beginnen.

6 Speicherverwaltung

6.1 Aufbau des Speichers

In C++ unterscheidet man vier Speicherbereiche:

- Im Bereich **Code** steht das Programm selbst.
- Der Bereich **Static** enthält alle globalen Variablen. Wir haben bislang nur Variablen gesehen, die in einer Funktion (und sei es die `main`-Funktion) definiert wurden. Prinzipiell (auch wenn dies nicht empfehlenswert ist) kann man aber Variablen auch so definieren, dass sie von jeder Funktion aus sichtbar sind. Wenn es sie gibt, stehen sie im Speicherbereich **Static**.
- Der **Stack** (Stapel) enthält lokale Variablen und Funktionsparameter, die dort nach dem LIFO(=last in first out)-Prinzip gespeichert werden. Eine Variable, deren Sichtbarkeitsbereich endet, wird automatisch aus dem Stack entfernt. Auch Speicherblöcke, deren Größe vor dem Programmablauf feststeht werden auf dem Stack abgelegt. Man muss sich um die Freigabe dieses Speichers als Programmierer nicht kümmern.

Darüber hinaus werden auf dem Stack Informationen darüber gespeichert, wohin der Programmfluss springen soll, wenn eine Funktion beendet wird.

- Der **Heap** (Halde) enthält Speicher, der zur Laufzeit dynamisch angelegt wird. Prinzipiell muss man diesen Speicher auch selbst wieder freigeben, wenn man ihn nicht mehr braucht, sonst ist im folgenden Programmablauf sinnlos blockiert. Eine automatische *Garbage Collection*, also das automatische Freigeben von nicht mehr erreichbarem Speicher, gibt es in C++ nicht (allerdings sogenannte Smart Pointer, die einen ähnlichen Effekt haben). Wir werden im folgenden Abschnitt aber sehen, wie man die Speicherreservierung im Heap trotzdem so gestalten kann, dass die Freigabe im Hintergrund geschieht, ohne dass wir selbst uns darum kümmern müssen.

6.2 Dynamische Speicherverwaltung mit der Klasse `vector`

6.2.1 Grundlegendes

Die C++-Standardbibliothek bietet einen recht komfortablen Weg an, sich Heap-Speicher zu holen, nämlich die Klasse `vector`. Was eine Klasse eigentlich ist, werden wir erst im nächsten Kapitel lernen, aber an dieser Stelle können wir uns schon anschauen, wie man eine Klasse verwendet. Das folgende Programm zeigt, wie man Speicher mit Hilfe von `vector` ersetzen kann.

```
1 #include <iostream>
2 #include <vector> // Liefert die Methoden zur Benutzung des Containers "vector"
3
4 int main()
5 {
6     std::cout << "Wie viele Zahlen wollen Sie eingeben? " << std::endl;
7     int zahl_der_eintraege;
8     std::cin >> zahl_der_eintraege;
9
10    if(zahl_der_eintraege < 0)
11    {
12        return 0;
13    }
14 }
```

```

15 // Lege vector an, der Platz fuer zahl_der_eintraege viele ints hat:
16 std::vector<int> eintraege(zahl_der_eintraege);
17
18 for(int i = 0; i < zahl_der_eintraege; i++)
19 {
20     std::cout << "Eintrag " << i << " bitte. ";
21     std::cin  >> eintraege[i];
22 }
23
24 std::cout << "Hier Ihre Zahlen: " << std::endl;
25
26 for(int i = 0; i < zahl_der_eintraege; i++)
27 {
28     std::cout << "Eintrag " << i << ": " << eintraege[i] << std::endl;
29 }
30
31 // Eine Speicherfreigabe ist hier nicht noetig.
32
33 return 0;
34 }

```

Listing 30: Programm, das gegebene Zahlen in einem Vektor speichert.

Initialisiert wird der Vektor mit der folgenden Zeile.

```

1 std::vector<int> eintraege(zahl_der_eintraege);

```

In den spitzen Klammern <> gibt man an, welchen Datentyp die Vektoreinträge haben sollen, in diesem Fall also `int`. In runden Klammern wird am Ende die geplante Länge des Vektors angegeben. Anschließend kann man mit eckigen Klammern `[]` auf die Einträge des Vektors zugreifen, um ihren Wert auszulesen oder zu überschreiben. Eine praktische Eigenschaft von `vector` ist, dass man den damit reservierten Speicher nicht selbst explizit freigeben muss. Wenn der Sichtbarkeitsbereich des Vektors endet, wird auch der Speicher freigegeben. Was dabei im Hintergrund passiert, sehen wir im folgenden Kapitel, wenn Destruktoren von Klassen erklärt werden.

Die Datenstruktur `vector` (ein sogenannter Container, weil er dazu dient, Daten zu speichern und zu verwalten) kann aber mehr. Betrachten wir dazu die folgende Version des vorigen Programms:

```

1 #include <iostream>
2 #include <vector> // Liefert die Methoden zur Benutzung des Containers "vector"
3
4 int main()
5 {
6     // Lege vector an, in dem ints gespeichert werden koennen:
7     std::vector<int> eintraege;
8
9     std::cout << "Geben Sie nicht-negative Zahlen ein. " << std::endl;
10    std::cout << "Mit einer negativen Zahl koennen Sie die Eingabe beenden" << std::endl;
11    int eingabe;
12
13    do
14    {
15        std::cout << "Naechster Eintrag bitte. ";
16        std::cin  >> eingabe;
17        if(eingabe >= 0)
18        {
19            eintraege.push_back(eingabe);
20        }
21    } while(eingabe >= 0);
22
23    std::cout << "Das waren " << eintraege.size() << " Zahlen" << std::endl;
24    std::cout << "Hier Ihre Zahlen: " << std::endl;
25
26    for(unsigned int i = 0; i < eintraege.size(); i++)
27    {
28        std::cout << "Eintrag " << i << ": " << eintraege[i] << std::endl;
29    }

```

```

30
31 // Eine Speicherfreigabe ist hier nicht noetig.
32
33 return 0;
34 }

```

Listing 31: Programm, das gegebene Zahlen in einem Vektor speichert, ohne dass deren Gesamtzahl vorher bekannt war.

In dieser Version gibt es drei wesentliche Unterschiede. Zum einen wird bei der Initialisierung keine Größe des geplanten Arrays angegeben:

```

1 std::vector<int> eintraege;

```

Die Gesamtzahl der Einträge steht ja auch anfangs gar nicht fest. Daher kann man nun natürlich auch nicht einfach mit `eintraege[i]` auf einen *i*-ten Eintrag zugreifen und ihn ändern. Die Datenstruktur `vector` verfügt aber über eine Möglichkeit, dynamisch ein Array um einen Eintrag zu erweitern. Diese geschieht hier mit folgendem Befehl:

```

1 eintraege.push_back(eintrag);

```

Dieser Befehl hängt an das Ende des Vektors ein Element, in dem der Wert von `eintrag` gespeichert wird an. Diese Syntax mit dem Punkt zwischen `eintraege` und `push_back` ist an dieser Stelle neu. Es handelt sich um eine Methode der *Klasse* `vector`. Mehr dazu werden wir im nächsten Kapitel sehen, im Moment reicht es, wenn wir wissen, dass man auf diesem Weg einen Vektor dynamisch erweitern kann, ohne sich selbst um den benötigten Speicher zu kümmern. Mit einer Methode namens `size()`, die wie `push_back()` zu einem Vektor-Objekt gehört, können wir einen Vektor nach der Zahl seiner Einträge fragen. Davon machen wir in den Zeilen

```

1 std::cout << "Das waren " << eintraege.size() << " Zahlen" << std::endl;

```

und

```

1 for(unsigned int i = 0; i < eintraege.size(); i++)

```

Gebrauch. Wir müssen uns also nicht explizit selbst merken, wie viele Einträge ein Vektor hat.

Man kann beim Anlegen eines Vektors auch gleich alle Einträge mit einem bestimmten Wert initialisieren, den man beim Erzeugen des Vektors als zweites Argument übergibt:

```

1 #include <iostream>
2 #include <vector> // Liefert die Methoden zur Benutzung des Containers "vector"
3
4 int main()
5 {
6     /*Legt einen double-Vektor der Laenge 3 an, dessen Eintraege
7      alle mit 2.7 initialisiert sind: */
8     std::vector<double> eintraege(3, 2.7);
9
10    for(unsigned int i = 0; i < eintraege.size(); i++)
11    {
12        std::cout << eintraege[i] << std::endl;
13    }
14    return 0;
15 }

```

Listing 32: Ein Vektor, dessen Einträge alle mit dem Wert 2.7 initialisiert werden.

Das obige Programm würde also dreimal den Wert 2.7 ausgeben.

Neben dem Zugriff mit eckigen Klammern `[]` auf ein Vektor-Element kann man auch mit der Methode `at()` auf ein solches zugreifen. In obigem Programm kann man Zeile 28 äquivalent durch folgende Zeile ersetzen:

```
std::cout << eintraege.at(i) << std::endl;
```

Der Vorteil der Syntax `eintraege.at(i)` gegenüber `eintraege[i]` ist, dass, wenn `i` außerhalb des zulässigen Bereichs ist (also größer als `eintraege.size()-1`, eine kontrollierte Fehlermeldung abgesetzt wird, während bei `eintraege[i]` das Verhalten schlicht undefiniert ist. Daher ist der Zugriff mit `eintraege.at(i)` generell von Vorteil.

6.2.2 Schleifen über Vektoren

Es gibt noch eine andere Möglichkeit, die Elemente eines Vektors in einer `for`-Schleife ohne explizite Nutzung einer Zählvariable zu durchlaufen, die im folgenden Beispiel gezeigt wird:

```
1 #include <iostream>
2 #include <vector> // Liefert die Methoden zur Benutzung des Containers "vector"
3
4 int main()
5 {
6     /*Legt einen double-Vektor der Laenge 3 an, dessen Eintraege
7      alle mit 2.7 initialisiert sind: */
8     std::vector<double> eintraege(3, 2.7);
9
10    for(double entry : eintraege) //Andere Syntax einer for-Schleife auf Vektoren
11    {
12        std::cout << entry << std::endl;
13    }
14    return 0;
15 }
```

Listing 33: Eine andere Art, Vektoren zu traversieren.

Dieses Programm macht dasselbe wie Listing 32, nur die Syntax für die Schleife wurde geändert in

```
1 for(double entry : eintraege)
```

Hiermit wird eine Schleife erzeugt, in der die Variable `entry` nacheinander alle Werte von Einträgen des Vektors `eintraege` annimmt (in der kanonischen Reihenfolge von `eintraege[0]` bis `eintraege[eintraege.size()-1]`). Die Variable `entry` ist hier natürlich vom Typ `double`, weil der betreffende Vektor Einträge vom Typ `double` hat. Weil die Variable nur vom Typ `double` sein kann, kann man es aber auch dem Compiler überlassen, herauszufinden, welchen Typ die Variable `entry` hat. Dazu gibt es das Schlüsselwort `auto` für eine Typ-Bezeichnung. In Listing 33 könnte man Zeile 10 durch folgenden Code ersetzt werden:

```
1 for(auto entry : eintraege)
```

Hier wird der Typ der Variablen `entry` nicht vom Nutzer spezifiziert. Der Compiler muss selbst erkennen, welcher Typ gewählt werden muss. Solange das eindeutig ist, kann man statt einer expliziten Typenangabe das Schlüsselwort `auto` verwenden.

6.2.3 Vektoren als Argumente von Funktionen

Es ist auch möglich, einen Vektor auch als Parameter an eine Funktion übergeben, wie in dem folgenden Beispiel:

```

1 #include <iostream>
2 #include <vector>
3
4 /* Obacht: Diese Funktion berechnet nicht nur die Summe
5    der Eintraege im Vektor zahlen, sondern setzt auch alle
6    Eintraege auf 37.*/
7 double summe(std::vector<double> zahlen)
8 {
9     double summe = 0;
10
11     for(unsigned int i = 0; i < zahlen.size(); i++)
12     {
13         summe += zahlen[i];
14         zahlen[i] = 37;
15     }
16     return summe;
17 }
18
19 int main()
20 {
21     /*Legt einen double-Vektor der Laenge 3 an, dessen Eintraege
22        alle mit 2.7 initialisiert sind: */
23     std::vector<double> eintraege(3, 2.7);
24
25     for(unsigned int i = 0; i < eintraege.size(); i++)
26     {
27         std::cout << "Eintrag (" << i << ") vorher: " << eintraege[i] << std::endl;
28     }
29
30     double gesamt = summe(eintraege);
31     std::cout << "Gesamt: " << gesamt << std::endl;
32
33     for(unsigned int i = 0; i < eintraege.size(); i++)
34     {
35         std::cout << "Eintrag (" << i << ") nachher: " << eintraege[i] << std::endl;
36     }
37
38     return 0;
39 }

```

Listing 34: Ein Vektor kann als Argument an eine Funktion übergeben werden.

Sinnloserweise wird in der Funktion `summe` nicht nur eine Summe berechnet, sondern alle betrachteten Einträge des übergebenen Vektors werden nach dem Aufsummieren auf 37 gesetzt. Auf den Vektor `eintraege` in `main()` hat diese Änderung aber keine Auswirkung, die Ausgabe des Programms sieht daher so aus:

```

Eintrag (0) vorher: 2.7
Eintrag (1) vorher: 2.7
Eintrag (2) vorher: 2.7
Gesamt: 8.1
Eintrag (0) nachher: 2.7
Eintrag (1) nachher: 2.7
Eintrag (2) nachher: 2.7

```

Der Vektor wird nämlich beim Funktionsaufruf kopiert und nur die Kopie bearbeitet, da wir den Vektor per Value und nicht per Referenz übergeben haben.

Das Kopieren eines Vektors wie in obigem Beispiel will man bei einem Funktionsaufruf aber eigentlich immer vermeiden. Der Vektor kann ja sehr groß werden, was man beim Implementieren unter Umständen noch gar nicht überblicken kann. Daher wäre die bessere Schnittstelle von folgender Form:


```
1 double summe(std::vector<double> & zahlen)
```

Allerdings hätte das hier die Konsequenz, dass das merkwürdige Überschreiben der Funktionswerte in der Funktion `summe` tatsächlich den Vektor `eintraege` aus der `main`-Funktion ändern würde.

```
Eintrag (0) vorher: 2.7
Eintrag (1) vorher: 2.7
Eintrag (2) vorher: 2.7
Gesamt: 8.1
Eintrag (0) nachher: 37
Eintrag (1) nachher: 37
Eintrag (2) nachher: 37
```

Dies ist aber natürlich nur ein unerwünschtes Verhalten in der Funktion `summe`. Verhindern kann man es mit dem Schlüsselwort `const`:

```
1 double summe(const std::vector<double> & zahlen)
```

Auf diese Weise wird der Vektor nicht kopiert, kann aber in der Funktion auch nicht verändert werden. Damit würde man also, wenn man doch versucht, Vektoreinträge zu verändern, einen Compiler-Fehler erhalten.

6.2.4 Vektoren als Rückgabe-Werte von Funktionen

Vektoren können auch als Rückgabe-Wert einer Funktion verwendet werden. Auf diese Weise kann man z.B. eine (im Prinzip) beliebig große Menge von Zahlen zurückgeben. Das folgende Programm zeigt ein Beispiel, in dem zu einer nicht-negativen ganzen Zahl `n` alle geraden nicht-negativen Zahlen, die höchstens `n` sind, in einem Vektor zurückgegeben werden.

```
1 #include <iostream>
2 #include <vector>
3
4 std::vector<unsigned int> gerade_zahlen(unsigned int n)
5 // Gibt einen Vektor zurueck, der alle gerade Zahlen von 0 bis n enthaelt.
6 {
7     unsigned int num_entries = n/2 + 1;
8     std::vector<unsigned int> ausgabe_vektor(num_entries);
9     for(unsigned int i = 0; i < num_entries; ++i)
10     {
11         ausgabe_vektor[i] = 2 * i;
12     }
13     return ausgabe_vektor;
14 }
15
16 int main()
17 {
18     std::cout << "Ein nicht-negative ganze Zahl bitte: " << std::endl;
19     unsigned int n = 0;
20     std::cin >> n;
21     std::vector<unsigned int> zahlen = gerade_zahlen(n);
22     std::cout << "Dies sind die geraden Zahlen bis " << n << ":" << std::endl;
23     for(unsigned int i = 0; i < zahlen.size(); i++)
24     {
25         std::cout << zahlen[i] << std::endl;
26     }
27     return 0;
28 }
```

Listing 35: Ein Vektor kann von einer Funktion zurückgegeben werden.

7 Klassen

7.1 Grundbegriffe

Die Möglichkeit, Klassen zu definieren, ist eine wesentliche Eigenschaft, die C++ von C unterscheidet. Eine Klasse fasst Daten und Methoden, um auf diesen Daten zu arbeiten, zu einem Objekt zusammen. Dadurch lassen sich komplexe Datentypen erzeugen, die aber in vieler Hinsicht wie elementare Datentypen wie etwa `int` oder `double` verwendet werden können.

Die Grundstruktur einer Klasse sieht so aus:

```
class KLASSENNAME {  
public:  
    // Eintraege, die von aussen sichtbar sein sollen  
    // (insbesondere ein oder mehrere Konstruktoren)  
private:  
    // Eintraege, die nicht von aussen sichtbar sein sollen  
};
```

Die Einträge sind jeweils entweder Variablen oder Funktionen.

Dazu ein Beispiel:

```
1 #include <iostream>  
2  
3 /* Sehr rudimentaere Version einer Klasse zur  
4    Verwaltung von Punkten in der Ebene: */  
5 class Punkt {  
6 public: // Oeffentlich zugaeungliche Daten  
7     Punkt(double x_coor, // Ein Constructor der Klasse  
8           double y_coor)  
9     {  
10         _x_coor = x_coor;  
11         _y_coor = y_coor;  
12     }  
13     double _x_coor;  
14     double _y_coor;  
15 }; // Ende der Beschreibung der Klasse  
16  
17 void punkt_ausgeben(const Punkt & punkt)  
18 {  
19     std::cout << "x: " << punkt._x_coor << " y: " << punkt._y_coor << std::endl;  
20 }  
21  
22 int main()  
23 {  
24     Punkt mein_punkt(37, 42.5);  
25  
26     punkt_ausgeben(mein_punkt);  
27     mein_punkt._y_coor = 2;  
28     punkt_ausgeben(mein_punkt);  
29     mein_punkt._x_coor += 15;  
30     punkt_ausgeben(mein_punkt);  
31 }
```

Listing 36: Eine sehr einfache Klasse zum Speichern von Punkten in der Ebene.

An diesem Beispiel wollen wir die wichtigsten syntaktischen Elemente im Umgang mit Klassen erklären.

Die Definition der Klasse beginnt mit dem Text `class Punkt`. Dabei ist `class` ein Schlüsselwort, das anzeigt, dass wir hier die Definition einer Klasse starten wollen. `Punkt` ist ein von uns gewählter Name der Klasse. Typischerweise schreibt man Klassennamen groß. Es folgt in geschweiften Klammern `{ }` die Definition der Klasse.

Der erste Klasseneintrag `public:` gibt an, dass man auf die folgenden Einträge zugreifen kann, wenn man ein Objekt von dieser Klasse zur Verfügung hat. In diesem Beispiel gilt dies für alle Einträge. Wir werden später sehen, dass es sinnvoll sein kann, Einträge vor dem Nutzer der Klasse zu “verstecken”.

Der nächste Eintrag ist ein sogenannter Konstruktor (oder *constructor*). Dies ist eine Funktion, die ebenso heißt wie die Klasse selbst, also in diesem Fall `Punkt`. Sie wird aufgerufen, wenn ein Objekt der Klasse erzeugt wird. Der Funktion werden in diesem Fall zwei `double`-Werte übergeben, die benutzt werden, um die beiden Koordinaten des Punktes zu initialisieren.

Die weiteren Einträge speichern den eigentlichen Inhalt der Klasse. Es sind zwei `doubles`, welche die Koordinaten des Punktes angeben. Die Variablennamen beginnen in diesem Beispiel alle mit einem Unterstrich “_”. Es ist nicht notwendig, die Variablen so zu benennen; für Variablen in Klassen sind alle Variablennamen erlaubt, die auch sonst möglich sind. Allerdings hat es sich sehr bewährt, Variablennamen von Klasseneinträgen stets mit einem Unterstrich zu beginnen, weil man dadurch sofort erkennt, welche Variablen Klasseneinträge sind.

Der Zugriff auf einen Klasseneintrag eines Objekts erfolgt mit der folgenden Syntax:

OBJEKTNAME.EINTRAGNAME

Wenn man in einem Objekt `punkt` der Klasse `Punkt` auf den Eintrag `_x_coor` eines Objekt zugreifen will, kann man das mit `punkt._x_coor` tun. In obigem Programm zeigt die Funktion `punkt_ausgeben` einige Beispiele. Wir haben auch schon bei der Klasse `vector` in Abschnitt 6.2 mit `push_back` und `size` Beispiele für einen Zugriff auf Klassenelemente gesehen. Man beachte, dass in dem Aufruf von `punkt_ausgeben` der Punkt per Referenz übergeben wird, weshalb er an dieser Stelle nicht kopiert wird.

Wie man ein Objekt von unserer Klasse erzeugt, ist in der ersten Zeile der `main`-Funktion zu sehen:

```
1 Punkt mein_punkt ( 37 , 42.5);
```

Dieser Befehl erzeugt ein Objekt `mein_punkt` von der Klasse `Punkt` und sorgt dafür, dass der Konstruktor aufgerufen wird, wodurch die einzelnen Klasseneinträge initialisiert werden.

In der `main`-Funktion sieht man auch zwei Beispiele, wie man Einträge in dieser Klasse modifizieren kann. Dazu kann man hier direkt auf die einzelnen Einträge zugreifen und ihre Werte wie die einer gewöhnlichen Variable abändern.

Wir schauen uns ein weiteres Beispiel an. Wir nehmen an, dass wir eine Datenstruktur haben wollen, in der wir achsenparallele Rechtecke mit ganzzahligen Koordinaten in der Ebene speichern wollen. Außerdem wollen wir den Flächeninhalt eines solchen Rechtecks immer abgespeichert haben, um ihn bei Bedarf ohne neue Berechnung abfragen zu können. Dies könnte, analog zu der Klasse `Punkt` mit einer Klasse, wie sie im folgenden Programm definiert wird, durchgeführt werden:

```
1 #include <iostream>
2
3 /* Sehr rudimentaere Version einer Klasse zur
4    Verwaltung von Rechtecken: */
5 class Rechteck_1 {
6 public:
7     Rechteck_1(int x_min, // Ein Constructor der Klasse
8               int x_max,
9               int y_min,
10              int y_max)
11 {
```

```

12     _x_min = x_min;
13     _x_max = x_max;
14     _y_min = y_min;
15     _y_max = y_max;
16     _flaecheninhalt = (_x_max - _x_min) * (_y_max - _y_min);
17 }
18 int _x_min;
19 int _x_max;
20 int _y_min;
21 int _y_max;
22 int _flaecheninhalt;
23 };
24
25 void rechteck_ausgeben(Recteck_1 &rechteck)
26 {
27     std::cout << "x: " << rechteck._x_min << " " << rechteck._x_max << std::endl;
28     std::cout << "y: " << rechteck._y_min << " " << rechteck._y_max << std::endl;
29     std::cout << "Flaecheninhalt: " << rechteck._flaecheninhalt << std::endl;
30 }
31
32 int main()
33 {
34     Recteck_1 mein_rechteck(0, 37, 10, 42);
35     rechteck_ausgeben(mein_rechteck);
36     mein_rechteck._x_min = 3; // Vorsicht: Dieser Befehl aendert die Koordinate, aber nicht _flaecheninhalt
37     rechteck_ausgeben(mein_rechteck);
38 }

```

Listing 37: Eine sehr einfache Klasse zum Speichern von Rechtecken in der Ebene.

Im Prinzip kann man Rechtecke so abspeichern und verwalten wie in Listing 37, allerdings ist diese Klasse fehleranfällig. In obigem Beispiel wird bei der Benutzung auch prompt ein Fehler gemacht. Der Wert `_x_min` wird auf 3 gesetzt, ohne dass der Flächeninhalt geändert wird. Die Einträge in der Klasse sind also nicht mehr konsistent, die Ausgabe dieses Programm sähe so aus.

```

x: 0 37
y: 10 42
Flaecheninhalt: 1184
x: 1 37
y: 10 42
Flaecheninhalt: 1184

```

Wenn man erzwingen will, dass die in der Klasse definierten Daten konsistent bleiben, muss man dafür sorgen, dass bei jeder Änderung einer Koordinate auch der Wert des Eintrags `flaecheninhalt` aktualisiert wird. Dazu kann man das Schlüsselwort **private** verwenden, das verhindert, dass auf bestimmte Einträge einer Klasse direkt zugegriffen wird.

```

1 #include <iostream>
2
3 /* Rudimentaere Version einer Klasse zur
4  Verwaltung von Rechtecken: */
5 class Recteck_2 {
6 public: // Oeffentlich zugaeengliche Daten
7     Recteck_2(int x_min, // Ein Constructor der Klasse
8               int x_max,
9               int y_min,
10              int y_max)
11     {
12         _x_min = x_min;
13         _x_max = x_max;
14         _y_min = y_min;
15         _y_max = y_max;
16         update_flaecheninhalt();
17     }

```

```

18 // Zugriffsfunktionen auf die Eintraege:
19 int get_x_min() const {
20     return _x_min;
21 }
22 int get_x_max() const {
23     return _x_max;
24 }
25 int get_y_min() const {
26     return _y_min;
27 }
28 int get_y_max() const {
29     return _y_max;
30 }
31 int get_flaecheninhalt() const {
32     return _flaecheninhalt;
33 }
34 // Funktionen zum Aendern der Eintraege:
35 void set_x_min(int x_min) {
36     _x_min = x_min;
37     update_flaecheninhalt();
38 }
39 void set_x_max(int x_max) {
40     _x_max = x_max;
41     update_flaecheninhalt();
42 }
43 void set_y_min(int y_min) {
44     _y_min = y_min;
45     update_flaecheninhalt();
46 }
47 void set_y_max(int y_max) {
48     _y_max = y_max;
49     update_flaecheninhalt();
50 }
51 private: // Nicht oeffentlich zugaeengliche Daten:
52     int _x_min;
53     int _x_max;
54     int _y_min;
55     int _y_max;
56     int _flaecheninhalt;
57     void update_flaecheninhalt() {
58         _flaecheninhalt = (_x_max - _x_min) * (_y_max - _y_min);
59     }
60 }; // Ende der Beschreibung der Klasse
61
62 void rechteck_ausgeben(const Rechteck_2 &rechteck)
63 {
64     std::cout << "x: " << rechteck.get_x_min() << " " << rechteck.get_x_max() << std::endl;
65     std::cout << "y: " << rechteck.get_y_min() << " " << rechteck.get_y_max() << std::endl;
66     std::cout << "Flaecheninhalt: " << rechteck.get_flaecheninhalt() << std::endl;
67 }
68
69 int main()
70 {
71     Rechteck_2 mein_rechteck(0, 37, 10, 42);
72     rechteck_ausgeben(mein_rechteck);
73     mein_rechteck.set_x_min(3);
74     rechteck_ausgeben(mein_rechteck);
75 }

```

Listing 38: Eine etwas erweiterte Klasse zum Speichern von Rechtecken in der Ebene.

Die Ausgabe dieses Programms sieht dann so aus:

```

x: 0 37
y: 10 42
Flaecheninhalt: 1184
x: 1 37
y: 10 42

```

Flaecheninhalt: 1088

Wenn man die Einträge der Klasse jetzt mit `set_x_min` usw. verändert, wird automatisch sichergestellt, dass der Eintrag `_flaecheninhalt` stets mit angepasst wird. Die Benutzung dieser Methoden ist aber auch die *einzigste* Art, die Klasseneinträge nach der Initialisierung noch einmal zu verändern. Denn mit der Klasse `Rechteck_2`, wie sie in Listing 38 eingeführt wird, würde jetzt folgendes Code-Fragment zu einem Compiler-Fehler führen:

```
1 Rechteck_2 mein_rechteck(0, 37, 10, 42);
2 mein_rechteck._x_min = 3; // Fehler: Dies geht mit der neuen Version von Rechteck nicht mehr.
```

Methoden innerhalb der Klasse (wie die Methoden `set_x_min` usw.) können aber natürlich auch auf alle Klasseneinträge zugreifen, die unter `private` stehen.

Man beachte, dass in der Klasse `Rechteck_2` bei den Methoden, deren Namen mit `get` beginnt, in der Definition das Schlüsselwort `const` steht, wie etwa bei `int get_x_min() const`. Das Schlüsselwort `const` zeigt an dieser Stelle an, dass durch die Methode kein Eintrag der Klasse verändert wird. Wenn doch eine Änderung in so einer Methode vorgenommen würde, dann gäbe es einen Fehler beim Kompilieren. Da prinzipiell alle Klassenmethoden alle Klasseneinträge ändern können, ist es für die Übersichtlichkeit sehr empfehlenswert, Methoden, die nichts ändern, als `const` zu kennzeichnen. Außerdem kann man dann in einer Funktion wie `rechteck_ausgeben` das Rechteck als *konstante* Referenz übergeben (so wie in Listing 38 geschehen). Wenn ein Klassenobjekt als konstante Referenz übergeben wird, hat man auch nur Zugriff auf die mit `const` markierten Methoden. Daher konnte bei der Version von `rechteck_ausgeben` in Listing 37 keine `const`-Referenz übergeben werden.

Das folgende Programm enthält eine weitere Version einer Rechteck-Klasse:

```
1 #include <iostream>
2
3 enum Dimension {x_dim, y_dim}; // Die moeglichen Dimensionen als enum
4
5 using coord_type = int; // Hier erhalten die Koordinaten einen eigenen Typennamen
6
7 class Rechteck_3 {
8 public: // Oeffentlich zugaeengliche Daten
9     Rechteck_3(coord_type x_min, // Ein Constructor der Klasse
10                coord_type x_max,
11                coord_type y_min,
12                coord_type y_max)
13     {
14         _x_min = x_min;
15         _x_max = x_max;
16         _y_min = y_min;
17         _y_max = y_max;
18         update_flaecheninhalt();
19     }
20     // Zugriffsfunktionen auf die Eintraege:
21     coord_type get_min(Dimension dimension) const {
22         if(dimension == x_dim)
23         {
24             return _x_min;
25         }
26         return _y_min;
27     }
28
29     coord_type get_max(Dimension dimension) const {
30         if(dimension == x_dim)
31         {
32             return _x_max;
33         }
34         return _y_max;
35     }
```

```

35 }
36
37 coor_type get_flaecheninhalt() const {
38     return _flaecheninhalt;
39 }
40 // Funktionen zum Aendern der Eintraege:
41 void set_min(coor_type value, Dimension dimension) {
42     if(dimension == x_dim)
43     {
44         _x_min = value;
45     }
46     _y_min = value;
47     update_flaecheninhalt();
48 }
49 void set_max(coor_type value, Dimension dimension) {
50     if(dimension == x_dim)
51     {
52         _x_max = value;
53     }
54     _y_max = value;
55     update_flaecheninhalt();
56 }
57 private: // Nicht oeffentlich zugaeengliche Daten:
58 coor_type _x_min;
59 coor_type _x_max;
60 coor_type _y_min;
61 coor_type _y_max;
62 coor_type _flaecheninhalt;
63 void update_flaecheninhalt() {
64     _flaecheninhalt = (_x_max - _x_min) * (_y_max - _y_min);
65 }
66 }; // Ende der Beschreibung der Klasse
67
68 void rechteck_ausgeben(const Rechteck_3 &rechteck)
69 {
70     std::cout << "x: " << rechteck.get_min(x_dim) << " " << rechteck.get_max(x_dim) << std::endl;
71     std::cout << "y: " << rechteck.get_min(y_dim) << " " << rechteck.get_max(y_dim) << std::endl;
72     std::cout << "Flaecheninhalt: " << rechteck.get_flaecheninhalt() << std::endl;
73 }
74
75 int main()
76 {
77     Rechteck_3 mein_rechteck(0, 37, 10, 42);
78     rechteck_ausgeben(mein_rechteck);
79     mein_rechteck.set_min(3, x_dim);
80     rechteck_ausgeben(mein_rechteck);
81 }

```

Listing 39: Eine erweiterte Klasse zum Speichern von Rechtecken in der Ebene.

Diese Version enthält mehrere Veränderungen. Zum einen wird dem Koordinaten-Typ ein eigener Name (`coor_type`) gegeben. Wenn also Variablen diesen Typ haben, sieht man ihnen sofort an, dass sie als Koordinaten zu interpretieren sind. Außerdem muss man nur eine Zeile ändern, wenn man z.B. statt `int` den Typ `double` benutzen will (wie es noch flexibler geht, sehen wir in Abschnitt 7.8).

Außerdem wird bei den lesenden und schreibenden Zugriffen auf die Koordinaten die Richtung als `x_dim` oder `y_dim` mitgegeben (codiert über ein `enum`). Das verkleinert die Zahl der Funktionen etwas, was schon deshalb gut ist, weil man weniger Code-Kopien hat. Wenn man die Klasse nun z.B. auf Punkte im dreidimensionalen Raum erweitern wollte, brauchte man keine zusätzlichen Funktionen.

7.2 Modularisierung mit Klassen

Man unterscheidet auch bei Klassen zwischen der Definition und der Deklaration. In der Deklaration stehen nur die Variablen und die Deklarationen der einzelnen Methoden. Dies erlaubt auch wieder eine Modularisierung, indem man die Deklaration in eine Header-Datei schreibt, während die Definition in einer cpp-Datei steht. Basierend auf der Klasse `Rechteck_3` aus Listing 39 kann die Header-Datei dann so aussehen:

```
1 #include <iostream>
2
3 class Rechteck_4 {
4
5 public:
6     enum Dimension {x_dim, y_dim};
7     using coor_type = int;
8
9     Rechteck_4(coor_type x_min,
10               coor_type x_max,
11               coor_type y_min,
12               coor_type y_max);
13     coor_type get_min(Dimension dim) const;
14     coor_type get_max(Dimension dim) const;
15     coor_type get_flaecheninhalt() const;
16     void set_min(coor_type coor, Dimension dim);
17     void set_max(coor_type coor, Dimension dim);
18 private:
19     coor_type _min[2];
20     coor_type _max[2];
21     coor_type _flaecheninhalt;
22     void update_flaecheninhalt();
23 };
```

Listing 40: Deklaration einer Rechteck-Klasse in einer Header-Datei.

Man beachte, dass wir hier zusätzlich die Definition von `Dimension` und `coor_type` in die Klasse hineingezogen haben. Das wäre für die Modularisierung nicht nötig gewesen, kann aber praktischer sein, wenn man z.B. an anderer Stelle ebenfalls `coor_type` für einen anderen Zweck benutzen möchte.

Die zugehörige cpp-Datei kann dann so aussehen:

```
1 #include <iostream>
2 #include "rechtecke.hpp"
3
4 Rechteck_4::Rechteck_4(coor_type x_min,
5                       coor_type x_max,
6                       coor_type y_min,
7                       coor_type y_max)
8 {
9     _min[x_dim] = x_min;
10    _max[x_dim] = x_max;
11    _min[y_dim] = y_min;
12    _max[y_dim] = y_max;
13    update_flaecheninhalt();
14 }
15
16 Rechteck_4::coor_type Rechteck_4::get_min(Dimension dim) const
17 {
18     return _min[dim];
19 }
20 Rechteck_4::coor_type Rechteck_4::get_max(Dimension dim) const
21 {
22     return _max[dim];
23 }
24 Rechteck_4::coor_type Rechteck_4::get_flaecheninhalt() const
25 {
26     return _flaecheninhalt;
27 }
```



```

28 void Rechteck_4::set_min(Rechteck_4::coord_type coord, Dimension dim)
29 {
30     _min[dim] = coord;
31     update_flaecheninhalt();
32 }
33 void Rechteck_4::set_max(Rechteck_4::coord_type coord, Dimension dim)
34 {
35     _max[dim] = coord;
36     update_flaecheninhalt();
37 }
38
39 void Rechteck_4::update_flaecheninhalt()
40 {
41     _flaecheninhalt = (_max[x_dim] - _min[x_dim]) * (_max[y_dim] - _min[y_dim]);
42 }

```

Listing 41: Zugehörige Definition der Klasseneinträge.

Hier müssen nun alle Methoden-Namen durch ein vorgeschaltetes `Rechteck_4::` ergänzt werden, damit deutlich wird, zu welcher Klasse die jeweilige Methode gehört. Ebenso muss man z.B. auf `coord_type` mit `Rechteck_4::coord_type` zugreifen.

Benutzt werden kann die Klasse dann wie folgt in einer `main`-Funktion:

```

1  #include <iostream>
2  #include "rechtecke.hpp"
3
4  void rechteck_ausgeben(const Rechteck_4 & rechteck)
5  {
6      std::cout << "x: " << rechteck.get_min(Rechteck_4::x_dim) << " "
7              << rechteck.get_max(Rechteck_4::x_dim) << std::endl;
8      std::cout << "y: " << rechteck.get_min(Rechteck_4::y_dim) << " "
9              << rechteck.get_max(Rechteck_4::y_dim) << std::endl;
10     std::cout << "Flaecheninhalt: " << rechteck.get_flaecheninhalt() << std::endl;
11 }
12
13 int main()
14 {
15     Rechteck_4 mein_rechteck(0, 37, 10, 42);
16     rechteck_ausgeben(mein_rechteck);
17     mein_rechteck.set_min(3, Rechteck_4::x_dim);
18     rechteck_ausgeben(mein_rechteck);
19 }

```

Listing 42: Benutzung der modularisierten Klasse in einer Hauptfunktion.

7.3 Weiteres zu Konstruktoren

Wenn Elemente einer Klasse im Konstruktor einfach durch gegebene Werte initialisiert werden sollen, kann man eine andere Syntax als die schon bekannte nutzen, die wir erst an einem Beispiel demonstrieren wollen. Der folgende Konstruktor kann den Konstruktor in Listing 38 äquivalent ersetzen:

```

1  Rechteck_2(int x_min,
2             int x_max,
3             int y_min,
4             int y_max):
5      _x_min(x_min),
6      _x_max(x_max),
7      _y_min(y_min),
8      _y_max(y_max)
9  {
10     update_flaecheninhalt();
11 }

```

An den Funktionskopf hängt man also einen Doppelpunkt an. Dann folgen, getrennt durch Kommata, mit der Syntax

`EINTRAGNAME(WERT)`

die einzelnen Initialisierungen.

Beide Konstruktoren werden auf dieselbe Art aufgerufen und beide Konstruktoren haben denselben Effekt für den Inhalt der Klasse. Dennoch gibt es einen subtilen Unterschied zwischen dem obigen Konstruktor und dem in Listing 38. In Listing 38 werden die einzelnen Einträge erst angelegt und irgendwie initialisiert, *danach* werden sie innerhalb des Konstruktors mit den eigentlichen Werten überschrieben. Die oben angegebene Syntax für den Konstruktor initialisiert die Einträge direkt mit den richtigen Werten. Dieser Unterschied spricht für die neue Version, da sie einen Schritt weniger benötigt. Allerdings wirkt sich dieser Unterschied in diesem konkreten Fall, in dem nur ein paar `int` überschrieben werden müssen, kaum aus.

Eine Klasse kann auch mehrere Konstruktoren haben. Beispielsweise könnte die Klasse `Rechteck_2` zusätzlich den folgenden Konstruktor haben:

```
1 Rechteck_2():
2     _x_min(0),
3     _x_max(0),
4     _y_min(0),
5     _y_max(0)
6 {
7     update_flaecheninhalt();
8 }
```

Damit wäre es nun möglich mit dem folgenden Code ein Rechteck zu erzeugen:

```
1 Rechteck_2 mein_rechteck;
```

Es ist aber zu beachten, dass der Compiler eine Chance haben muss, jeweils den richtigen Konstruktor zu finden. Deswegen müssen sich die verschiedenen Konstruktoren in ihren Schnittstellen unterscheiden. Man kann also nicht zusätzlich dem in Listing 38 definierten Konstruktor einen Konstruktor der folgenden Form haben:

```
1 Rechteck_2(int x_min,
2             int width,
3             int y_min,
4             int height):
5     _x_min(x_min),
6     _x_max(x_min + width),
7     _y_min(y_min),
8     _y_max(y_min + height)
9 {
10     update_flaecheninhalt();
11 }
```

Dieser Konstruktor wäre für sich genommen vollkommen korrekt, jedoch kann er nicht zusammen mit dem Konstruktor aus Listing 38 existieren, da der Compiler dann nicht weiß, wie er `Rechteck_2 mein_rechteck (0, 37, 10, 42);` interpretieren soll.

Abschließend sei noch erwähnt, dass eine Klasse auch ganz ohne Konstruktor auskäme. Allerdings sind dann die Werte in der Klasse erstmal nicht initialisiert, weshalb das typischerweise nicht zu empfehlen ist.

7.4 Destruktoren

Zu jeder Klasse gibt es einen Destruktor. So wie ein Konstruktor einer Klasse aufgerufen wird, wenn ein Objekt dieser Klasse initialisiert wird, wird der Destruktor aufgerufen, wenn das Objekt wieder verschwindet. Der Name des Destruktor entsteht aus dem Klassennamen, dem eine Tilde ~ vorangestellt wird, also heißt z.B. bei der Klasse `Punkt` der Destruktor `~Punkt()`. Man muss aber nicht explizit einen Destruktor angeben. Wenn man keinen spezifiziert, wird ein Default-Destruktor verwendet, der auch, falls vorhanden, die Destruktoren von möglichen Objekten innerhalb der Klasse aufruft (Klassen können auch Klassenobjekte enthalten, dazu haben wir aber noch kein Beispiel gesehen).

Im allgemeinen wird man mit als Anwender mit den Standard-Destruktoren auskommen, und Sie werden selbst keine Destruktoren schreiben müssen. Wir werden aber im nächsten Abschnitt ein kleines Beispiel sehen, wie Destruktoren zu Statistik-Zwecken verwendet werden können.

Ein Destruktor wird z.B. auch in der Klasse `vector` verwendet, um den darin reservierten Speicher wieder freizugeben, sobald der Sichtbarkeitsbereich der zugehörigen Variable endet.

7.5 static-Variablen

Das Schlüsselwort `static` vor einer Variablen in einer Klasse gibt an, dass diese Variable für alle Instanzierungen der Klasse, also für alle Objekt des Klassentyps dieselbe ist. Eine solche Variable wird global initialisiert und kann durch Konstruktoren, Destruktoren oder andere Klassenmethoden verändert werden (dann aber eben jedesmal für alle Objekte von dem Klassentyp. Hier ist ein Beispiel, in dem eine `static`-Variable `counter` mitzählt,

```
1 #include <iostream>
2
3 enum Dimension {x_dim, y_dim};
4
5 using coor_type = int;
6
7 class Rechteck_5 {
8 public:
9     Rechteck_5(coor_type x_min,
10                coor_type x_max,
11                coor_type y_min,
12                coor_type y_max)
13     {
14         _min[x_dim] = x_min;
15         _max[x_dim] = x_max;
16         _min[y_dim] = y_min;
17         _max[y_dim] = y_max;
18         update_flaecheninhalt();
19         _counter++; // Ein neues Rechteck wird erzeugt, also wird counter hochgezaehlt.
20     }
21     ~Rechteck_5()
22     {
23         _counter--; // Ein Rechteck verschwindet, also wird counter verkleinert.
24     }
25     coor_type get_min(Dimension dim) const {
26         return _min[dim];
27     }
28     coor_type get_max(Dimension dim) const {
29         return _max[dim];
30     }
31     coor_type get_flaecheninhalt() const {
32         return _flaecheninhalt;
33     }
34     void ausgeben()
35     {
36         std::cout << "[" << _min[x_dim] << ", " << _max[x_dim] << "] x "
```

```

37         << "[" << _min[y_dim] << ", " << _max[y_dim] << "; "
38         << "Flaecheninhalt: " << _flaecheninhalt << std::endl;
39         std::cout << "Counter: " << _counter << std::endl;
40     }
41     void set_min(coor_type coor, Dimension dim) {
42         _min[dim] = coor;
43         update_flaecheninhalt();
44     }
45     void set_max(coor_type coor, Dimension dim) {
46         _max[dim] = coor;
47         update_flaecheninhalt();
48     }
49 private:
50     coor_type _min[2];
51     coor_type _max[2];
52     coor_type _flaecheninhalt;
53     void update_flaecheninhalt() {
54         _flaecheninhalt = (_max[x_dim] - _min[x_dim]) * (_max[y_dim] - _min[y_dim]);
55     }
56     static int _counter; // Dieser Zaehler ist static, d.h. es ist zu jedem Zeitpunkt fuer
57                          // alle Variablen vom Typ Rechteck_5 dieselbe Variable
58 };
59
60 int Rechteck_5::_counter = 0; // Hier wird der Wert von _counter auf 0 gesetzt.
61
62 int main()
63 {
64     Rechteck_5 mein_rechteck(0, 37, 10, 42);
65     mein_rechteck.ausgeben();
66     mein_rechteck.set_min(3,x_dim);
67     mein_rechteck.ausgeben();
68     Rechteck_5 mein_rechteck2(2, 7, 11, 42);
69     mein_rechteck2.ausgeben();
70     {
71         Rechteck_5 mein_rechteck3(4, 7, 11, 42);
72         mein_rechteck3.ausgeben();
73     }
74     mein_rechteck2.ausgeben();
75     Rechteck_5 mein_rechteck4(2, 7, 11, 42);
76     mein_rechteck2.ausgeben();
77 }

```

Listing 43: Beispiel für eine static-Variable in einer Klasse.

Die Ausgabe sieht dann so aus:

```

[0, 37] x [10, 42]; Flaecheninhalt: 1184
Counter: 1
[3, 37] x [10, 42]; Flaecheninhalt: 1088
Counter: 1
[2, 7] x [11, 42]; Flaecheninhalt: 155
Counter: 2
[4, 7] x [11, 42]; Flaecheninhalt: 93
Counter: 3
[2, 7] x [11, 42]; Flaecheninhalt: 155
Counter: 2
[2, 7] x [11, 42]; Flaecheninhalt: 155
Counter: 3

```

7.6 Objekte als Klassenelemente

Um Rechtecke in beliebigen Dimensionen zu verwalten, kann man die Koordinaten in Vektoren abspeichern. Das folgende Programm zeigt ein Beispiel.

```
1 #include <iostream>
2 #include <vector>
3
4 /* Eine Klasse, die Rechtecke in beliebigen Dimensionen verwaltet. */
5 class Rechteck_Multi {
6 public:
7     Rechteck_Multi(int num_coors)
8     //Konstruktor, der einen num_coors-dimensionalen Einheitswürfel anlegt.
9     {
10         _min_coors = std::vector<double>(num_coors,0);
11         _max_coors = std::vector<double>(num_coors,1);
12     }
13     int get_num_coors() const {
14         return _min_coors.size();
15         //In der Annahme, dass dies immer gleich _min_coors.size() ist
16     }
17     double get_min_coor(int index) const {
18         return _min_coors[index];
19     }
20     double get_max_coor(int index) const {
21         return _max_coors[index];
22     }
23     void set_min_coor(int index, double value)
24     {
25         _min_coors[index] = value;
26     }
27     void set_max_coor(int index, double value)
28     {
29         _max_coors[index] = value;
30     }
31 private:
32     std::vector<double> _min_coors;
33     std::vector<double> _max_coors;
34 };
35
36 void rechteck_ausgeben(const Rechteck_Multi & rechteck)
37 {
38     for(int i = 0; i < rechteck.get_num_coors() ; i++)
39     {
40         std::cout << i << " : (" << rechteck.get_min_coor(i) << ", "
41             << rechteck.get_max_coor(i) << ")" << std::endl;
42     }
43     std::cout << std::endl;
44 }
45
46 int main()
47 {
48     Rechteck_Multi mein_rechteck(4);
49     rechteck_ausgeben(mein_rechteck);
50     Rechteck_Multi mein_anderes_rechteck = mein_rechteck;
51     mein_rechteck.set_max_coor(3, 2.0);
52     rechteck_ausgeben(mein_rechteck);
53     rechteck_ausgeben(mein_anderes_rechteck);
54 }
```

Listing 44: Eine Klasse, die beliebig-dimensionale Rechtecke verwalten kann.

Hier enthält die Klasse zwei Vektoren, einen für die minimalen Koordinaten und einen für die maximalen Koordinaten. Diese werden im Konstruktor angelegt und im (hier wie üblich nicht explizit spezifizierten) Destruktor wieder freigegeben.

7.7 Operatoren

Bei einigen Klassenobjekten bietet es sich an, arithmetische Operationen auf diesen Objekten zu definieren. Beispielsweise könnte man die Addition von Punkten im \mathbb{R}^2 als komponentenweise Addition definieren. Das wäre möglich, indem man in der Klasse `Punkt` die folgende Methode definiert:

```
1 Punkt addieren (const Punkt & punkt) const
2 {
3     return Punkt ( _x_coor + punkt._x_coor,
4                   _y_coor + punkt._y_coor);
5 }
```

Damit könnte man wie folgt die Summe von zwei Punkten `p1` und `p2` berechnen und als Punkt `p3` abspeichern:

```
1 Punkt p1 (37 , 42.5);
2 Punkt p2 (21 , 753);
3
4 Punkt p3 = p1.addieren(p2);
```

Hübscher wäre aber zweifellos die Schreibweise `p3 = p1 + p2`; Diese können wir benutzen, wenn wir, wie im folgenden Programm den `+`-Operator für Punkte definieren:

```
1 #include <iostream>
2
3 /* Sehr rudimentaere Version einer Klasse zur
4    Verwaltung von Punkten in der Ebene: */
5 class Punkt {
6 public: // Oeffentlich zugaeengliche Daten
7     Punkt(double x_coor, // Ein Constructor der Klasse
8           double y_coor)
9     {
10         _x_coor = x_coor;
11         _y_coor = y_coor;
12     }
13     double _x_coor;
14     double _y_coor;
15
16     Punkt operator+ (const Punkt & punkt) const
17     {
18         return Punkt(_x_coor + punkt._x_coor,
19                     _y_coor + punkt._y_coor);
20     }
21 }; // Ende der Beschreibung der Klasse
22
23 void punkt_ausgeben(const Punkt & punkt)
24 {
25     std::cout << "x: " << punkt._x_coor << " y: " << punkt._y_coor << std::endl;
26 }
27
28 int main()
29 {
30     Punkt p1(37, 42.5);
31     Punkt p2(21, 753);
32
33     punkt_ausgeben(p1);
34     punkt_ausgeben(p2);
35
36     Punkt p3 = p1 + p2;
37     punkt_ausgeben(p3);
38 }
39 }
```

Listing 45: Ein Operator für die Addition von zwei Punkten.

Hier wird mit

```

1 Punkt operator+ (const Punkt & punkt) const
2 {
3     return Punkt(_x_coor + punkt._x_coor,
4                 _y_coor + punkt._y_coor);
5 }

```

eine Klassenmethode definiert. Man könnte nun diese Methode theoretisch mit der Syntax `p3 = p1.operator+(p2);` benutzen. Mit der speziellen Syntax `operator+` ist dann aber auch die gewünschte Schreibweise `p3 = p1 + p2;` möglich. Analog kann man `operator-` definieren. Wenn man die Punkte als komplexe Zahlen interpretiert, ist es beispielsweise auch möglich, eine Multiplikation `operator*` zu definieren.

Schön wäre es auch, wenn man ein Klassenobjekt einfach über `std::cout` ausgeben lassen könnte. Auch das ist mit der Syntax wie in Programm 46 möglich. Dort wird der Operator `std::ostream & operator<<(std::ostream & os, const Bruch & b)` definiert, der den Output-Stream `os` und eine konstante Referenz auf einen Bruch bekommt und (nach der Ausgabe des Punktes) auch einen solchen Output-Stream zurückgibt.

```

1 #include <iostream>
2
3 class Bruch {
4 public:
5     Bruch(int zaehler,
6           int nenner) {
7         _zaehler = zaehler;
8         _nenner = nenner;
9     }
10
11     void print() {
12         std::cout << _zaehler << " / " << _nenner << std::endl;
13     }
14
15     Bruch multiplizieren (const Bruch & bruch) const {
16         return Bruch(_zaehler * bruch.get_zaehler(),
17                     _nenner * bruch.get_nenner());
18     }
19
20     int get_zaehler() const {
21         return _zaehler;
22     }
23
24     int get_nenner() const {
25         return _nenner;
26     }
27
28 private:
29     int _zaehler;
30     int _nenner;
31 };
32
33 std::ostream & operator<<(std::ostream & os, const Bruch & b) {
34     os << b.get_zaehler() << "/" << b.get_nenner();
35     return os;
36 }
37
38 int main() {
39     Bruch a(37, 42);
40     Bruch b(21, 753);
41     Bruch c = a.multiplizieren(b);
42
43     std::cout << "c ist " << c << "." << std::endl;
44
45     a.print();
46     b.print();
47     c.print();

```

Listing 46: Beispiel für die Umdefinition des Outputoperators.

7.8 Templates

In Listing 39 haben wir gesehen, wie man die Klasse der Rechtecke so gestalten kann, dass man den Koordinatentyp mit einer Änderung in nur einer Zeile wechseln kann. Dennoch müssen wir uns für einen Typ entscheiden. Wenn wir sowohl eine Klasse von Rechtecken mit `int`-Koordinaten als auch eine mit `double`-Koordinaten haben wollen, bleibt uns mit diesem Verfahren nichts übrig, als zwei Klassen zu erzeugen, die sich nur dadurch unterscheiden, dass in der einen bei den Koordinaten überall `int` steht, während in der anderen jeweils `double` steht. Die Methoden der Klassen unterscheiden sich sonst nicht. Solch einen doppelten Code sollte man stets vermeiden, und C++ bietet auch eine bequeme Möglichkeit dazu, nämlich die *Templatisierung*. Diese erfolgt mit Hilfe der Schlüsselwörter `template` und `typename`. Dazu muss man die erste Zeile der Klassendefinition nur nach dem folgenden Schema abändern.

```
template <typename TYPENAME> class KLASSENNAME {
public:
    // Eintraege, die von aussen sichtbar sein sollen
    // (insbesondere ein oder mehrere Konstruktoren)
private:
    // Eintraege, die nicht von aussen sichtbar sein sollen
};
```

Anschließend kann man in der Klasse den gewählten Typennamen wie einen gewöhnlichen Datentyp verwenden. Für die Rechtecke kann das so aussehen:

```
1 #include <iostream>
2
3 enum Dimension {x_dim, y_dim};
4
5 //Die Klasse Rechteck_4 erhaelt Template-Parameter:
6 template <typename T> class Rechteck_4 {
7 public:
8     Rechteck_4(T x_min, //T kann nun wie ein gewoehnlicher Datentyp benutzt werden.
9                T x_max,
10               T y_min,
11               T y_max)
12     {
13         _min[x_dim] = x_min;
14         _max[x_dim] = x_max;
15         _min[y_dim] = y_min;
16         _max[y_dim] = y_max;
17         update_flaecheninhalt();
18     }
19     // Zugriffsfunktionen auf die Eintraege:
20     T get_min(Dimension dim) const {
21         return _min[dim];
22     }
23     T get_max(Dimension dim) const {
24         return _max[dim];
25     }
26     T get_flaecheninhalt() const {
27         return _flaecheninhalt;
28     }
29     // Funktionen zum Aendern der Eintraege:
30     void set_min(T coor, Dimension dim) {
31         _min[dim] = coor;
```



```

32     update_flaecheninhalt();
33 }
34 void set_max(T coor, Dimension dim) {
35     _max[dim] = coor;
36     update_flaecheninhalt();
37 }
38 /* Die folgende Methode muss T nicht kennen (solange man Variablen
39 vom Typ T mit std::cout ausgeben kann). */
40 void ausgeben()
41 {
42     std::cout << "x: " << _min[x_dim] << " " << _max[x_dim] << std::endl;
43     std::cout << "y: " << _min[y_dim] << " " << _max[y_dim] << std::endl;
44     std::cout << "Flaecheninhalt: " << _flaecheninhalt << std::endl;
45 }
46 private:
47     T _min[2];
48     T _max[2];
49     T _flaecheninhalt;
50 void update_flaecheninhalt() {
51     _flaecheninhalt = (_max[x_dim] - _min[x_dim]) * (_max[y_dim] - _min[y_dim]);
52 }
53 }; // Ende der Beschreibung der Klasse
54
55 /* Fuer die folgende Funktion muss der Typ T aber spezifiziert sein,
56 was hier durch "<int>" passiert. */
57 void rechteck_ausgeben(const Rechteck_4<int> & rechteck)
58 {
59     std::cout << "x: " << rechteck.get_min(x_dim) << " " << rechteck.get_max(x_dim) << std::endl;
60     std::cout << "y: " << rechteck.get_min(y_dim) << " " << rechteck.get_max(y_dim) << std::endl;
61     std::cout << "Flaecheninhalt: " << rechteck.get_flaecheninhalt() << std::endl;
62 }
63
64 int main()
65 {
66     /* Hier wird ein Rechteck mit int-Koordinaten erzeugt: */
67     Rechteck_4<int> mein_rechteck(0, 37, 10, 42);
68     rechteck_ausgeben(mein_rechteck);
69     mein_rechteck.set_min(3, x_dim);
70     mein_rechteck.ausgeben();
71
72     /* Und hier wird ein Rechteck mit double-Koordinaten erzeugt: */
73     Rechteck_4<double> mein_anderes_rechteck(3, 7.2, -1, 12.2);
74     mein_anderes_rechteck.ausgeben();
75 }

```

Listing 47: Eine templatisierte Klasse zum Speichern von Rechtecken in der Ebene.

TYPENAME ist in diesem Beispiel einfach T. Außerhalb der Klasse muss der Typ T spezifiziert werden, was mit spitzen Klammern <> passiert. Will man ein Rechteck erzeugen, in dem die Koordinaten vom Typ int sind, geschieht das so:

```

1 Rechteck_4<int> mein_rechteck (0, 37, 10, 42);

```

Wenn man nun Rechtecke in Schnittstellen von Funktionen verwenden will, muss man T ebenfalls spezifizieren. In dem Beispiel geschieht das in der Funktion `rechteck_ausgeben`. Diese ist nun nur für Rechtecke mit `int`-Koordinaten aufrufbar. Wenn man hier also nicht für jeden Datentyp eine eigene Version schreiben will, empfiehlt es sich, diese Funktion als Methode in die Klasse einzubinden. Dies geschieht in dem Beispiel durch die Methode `ausgeben()`. Diese funktioniert als Klassenmethode auch z.B. für Rechtecke mit `double`-Koordinaten. Daher ist

```

1 mein_anderes_rechteck.ausgeben();

```

in obigem Beispiel korrekt, während

```

1 rechteck_ausgeben(mein_anderes_rechteck); //Fehler: mein_anderes_rechteck hat double-Koordinaten

```

nicht funktionieren würde.

Natürlich muss man darauf achten, dass wenn man einen Template-Parameter spezifiziert, alle Methoden in der Klassen auch aus dem gewählten Datentyp definiert sind. In unserem Beispiel eignen sich für `T` z.B. nur Datentypen, für die die Subtraktion, die Multiplikation und die Ausgaben mit `std::cout` definiert sind. Wenn aber diese Voraussetzungen erfüllt sind, kann man auch beliebige komplexe Datentypen verwenden.

Templatisierungen werden in vielen Klassen, die von der Standardbibliothek zu Verfügung gestellt werden, verwendet. Ein Beispiel haben wir in der Klasse `std::vector` gesehen. Der Datentyp der Einträge wurde hier in spitzen Klammern spezifiziert. Ein weiteres Beispiel ist `static_cast`, das für viele verschiedene Datentypen funktioniert.

Noch ein Beispiel zeigt das folgende Programm:

```
1 #include <iostream>
2 #include <limits> // Header, der std::numeric_limits enthaelt.
3
4 int main()
5 {
6     std::cout << "Kleinstes int: " << std::numeric_limits<int>::min() << std::endl;
7     std::cout << "Groesstes int: " << std::numeric_limits<int>::max() << std::endl;
8
9     std::cout << "Kleinstes double: " << std::numeric_limits<double>::min() << std::endl;
10    std::cout << "Groesstes double: " << std::numeric_limits<double>::max() << std::endl;
11    std::cout << "Digits double: " << std::numeric_limits<double>::digits << std::endl;
12
13    std::cout << "Kleinstes long double: " << std::numeric_limits<long double>::min() << std::endl;
14    std::cout << "Groesstes long double: " << std::numeric_limits<long double>::max() << std::endl;
15    std::cout << "Digits long double: " << std::numeric_limits<long double>::digits << std::endl;
16
17    return 0;
18 }
```

Listing 48: Templatisierung beim Zugriff auf Eigenschaften von elementaren Datentypen.

Um Eigenschaften von elementaren Datentypen wie `int` und `double` auszugeben, stellt die Standardbibliothek `std::numeric_limits` bereit, das über Templatisierung für viele Datentypen verwendbar ist. Das obige Programm zeigt einige Beispiele von möglichen Funktionen, die weitgehend selbsterklärend sind.

Literatur

- Breymann, U. [2012]: *Der C++ Programmierer. C++ lernen - Professionell anwenden - Lösungen nutzen.* Hanser Verlag, 2012.
- Hougardy, S. und Vygen, J. [2018]: *Algorithmische Mathematik.* 2. Auflage, Springer, 2018.
- Koenig, A., Moo, B.E. [2003]: *Intensivkurs C++: Schneller Einstieg über die Standardbibliothek.* Addison-Wesley, 2003.
- Louis, D. [2018]: *C++. Das komplette Starterkit für den einfachen Einstieg in die Programmierung.* 2. Auflage, Hanser, 2018.
- Scheinerman, E. [2006]: *C++ for Mathematicians. An Introduction for Students and Professionals.* Chapman and Hall, 2006.
- Stroustrup, B. [2013]: *The C++ Programming Language.* 4. Auflage, Addison-Wesley, 2013.
- Stroustrup, B. [2014]: *Programming. Principles and Practice Using C++.* 2. Auflage, Addison-Wesley, 2014.
- Stroustrup, B. [2023]: *Eine Tour durch C++. Der praktische Leitfaden für modernes C++.* Übersetzung der 3. Auflage, mitp, 2023.
- Theis, T. [2020]: *Einstieg in C++.* Rheinwerk Verlag, 2020
- Wolf, J., Guddat, M. [2022]: *Grundkurs C++.* 4. Auflage, Rheinwerk Verlag, 2022.

Index

and, 27
at(), 47
auto, 47

bool, 21
break, 30

Casting, 23
char, 22
class, 50
clock(), 32
Code (Speicherbereich), 44
Compiler, 7
const, 25, 37
constructor, 51
Container, 45
continue, 30
cstdlib, 32
ctime, 32

Datentyp, 21
destructor, 59
Destruktor, 59
do...while, 28
double, 21

else, 26
enum, 30
Expression, 23

Fibonacci-Zahlen, 35
float, 22
for, 28
Funktion, 33
Funktionsdeklaration, 37
Funktionskopf, 38

Geometrische Reihe, 28

Headerdatei, 8, 39
Heap (Speicherbereich), 44

if, 26
Initialisierung, 21
int, 20
iostream, 8

Kommentare, 20
kompilieren, 6
Konstruktor, 51

Linker, 39
long double, 22
long int, 22

main(), 5
Makro, 41
Maschinencode, 7
Modulo-Operator, 24

namespace, 42
not, 27

Objektdatei, 39
Operator, 23, 62
or, 27
Output-Stream, 63

Präprozessor, 41
private, 52
public, 51
push_back(), 46

rand(), 32
RAND_MAX, 32
Referenz, 36
Register, 6
return, 33

Scope, 29
short int, 22
Sichtbarkeitsbereich, 29
size(), 46
srand(), 32
Stack, 44
Standardbibliothek, 7
static (Klassenvariable), 59
Static (Speicherbereich), 44
static_cast, 24
std::cin, 25
std::cout, 22
std::numeric_limits, 66
std::vector, 44

template, 64
typename, 64
Typenumwandlung, 23

unsigned, 22
using, 31

Variable, 20
void, 34

while, 28