Modernes C++ für Programmierer

Eine praxisorientierte Anleitung

Günter Kolousek

Dezember 2014

Vorwort

Die Idee zu diesem Buch kam mir, da ich kein Buch gefunden habe, das einem Programmierer bei seinem Einstieg in die produktive Programmierung mit C++11 unterstützt.

Das erste Mal habe ich mich ca. vor 25 Jahren mit C++ beschäftigt und etliche Jahre mit dieser Programmiersprache entwickelt. Dann wurden meine Hauptwerkzeuge für die Entwicklung hauptsächlich Java und Python. Nach einer langen Periode in der Entwicklung von verteilten Systemen bin ich vor einiger Zeit wieder mit C++ und im speziellen mit C++11 in Berührung gekommen. Und da hat sich *einiges* geändert.

Wie beginnt man am Besten mit einer Sprache, die man schon kennt, aber die sich stark verändert hat. C++11 fühlt sich im Vergleich zu C++ aus den 80er Jahren wie eine *neue* Programmiersprache an. Also stand ich da als Programmierer mit 30 Programmiererfahrung und hatte eine neue Programmiersprache vor mir.

Also habe ich jede Menge Bücher gekauft, durchgearbeitet und sehr viel programmiert. Wäre ich mit entsprechenden Unterlagen nicht schneller produktiv gewesen? Ja, ich denke schon. In diesem Sinne hoffe ich, dass dieses Buch meine Leser schnell an Ihr Ziel bringt, C++ produktiv zu verwenden!

Neunkirchen, im September 2014
Günter Kolousek

Über dieses Buch

Was ist die Zielsetzung dieses Buches?

Das Ziel dieses Buches ist es, den Leser *schnell* in die *moderne* Programmierung mittels C++ einzuführen, sodass dieser *produktiv* Programme entwickeln kann.

Da der Zweck dieses Buches ist, den Leser schnell in die Programmierung von modernem C++ einzuführen, werden die *wesentlichen* Teile von C++ beschrieben. Das bedeutet lediglich, dass diejenigen Teile weggelassen werden, die für die moderne Entwicklung mit C++ nicht oder nur selten benötigt werden.

Da C++ eine Programmiersprache ist, deren Wurzeln in die 70er Jahre des letzten Jahrhunderts zurückgehen und die permament weiterentwickelt wird, hat die Sprache mittlerweile einen großen Umfang und eine hohe Komplexität erreicht. Durch neue Elemente, die hauptsächlich im Jahr 2011 in die Programmiersprache als C++11 eingeflossen sind, wurden Möglichkeiten geschaffen *einfacher* zu programmieren, sodass viele ältere syntaktischen Elemente nicht mehr benötigt werden. Auch können Programmiertechniken, die früher essentiell gewesen sind, heute durch andere, einfachere Konstrukte ersetzt werden.

Aus diesem Grund wird auch nicht auf die Maschinen-nahe Programmierung eingegangen, die gänzlich andere Anforderungen hat als die Programmierung von Entwicklung von Programmen für den Endbenutzer oder für server-seitige Programme. In diesem Sinne behandelt dieses Buch high-level C++!

Das Buch soll also als eine *gut lesbare* Anleitung zum Erlernen der wesentlichen Teile von C++11 dienen, sodass schnell Programme mittels C++11 erstellt werden können, die produktiv eingesetzt werden können.

Für wen ist dieses Buch geschrieben?

Dieses Buch habe ich für Personen geschrieben, die schon **grundlegende** Programmierkenntnisse besitzen. Damit bezeichne ich Personen, die in einer objektorientierten Programmiersprache einfache Programme programmieren können.

Da es sich bei diesem Buch um *kein* Buch für Programmieranfänger handelt, bitte ich Anfänger in der Programmierung zuerst ein anderes Buch zur Hand zu nehmen, das die grundlegenden Begriffe, die Konzepte der Programmierung und die tatsächliche Programmierung vermittelt. Dann kommen Sie wieder zu diesem Buch zurück!

Es kommt es eigentlich nicht darauf an, in welcher Pogrammsprache die Programmiererfahrungen vorhanden sind, wichtiger ist vielmehr, dass Programme mit Ein- und Ausgabe, (lokalen und globalen) Variablen, Datentypen, Verzweigungen, Schleifen, Funktionen, Exceptions, Klassen mit Methoden und Vererbung erstellt werden können. Grundlegende Begriffe wie z.B. Algorithmus (eine Beschreibung des Lösungsweges eines Problems), der Begriff eines Prozesses (gestartetes Programm) oder Zahlensysteme sollten ebenfalls geläufig sein.

Wie ist dieses Buch aufgebaut?

Das Buch ist in Kapitel gegliedert, die jeweils in bestimmte Themen einführen:

Kapitel 1 Im ersten Kapitel wird Grundlegendes zu `C++` erklärt. Hier geht es darum ein Gefühl für die Art dieser Programmiersprache zu erhalten und auch ein wenig über die Bedeutung dieser verbreitenden Programmiersprache zu erahnen. Außerdem wird kurz angerissen wie die Infrastruktur von `C++` aussieht und wie man mit `C++` prinzipiell umgeht.

Kapitel 2 Dieses Kapitel liefert einen Schnelleinstieg in die prozedurale Programmierung mit `C++`. Hier findet sich ein Überblick über den grundlegenden Aufbau eines einfachen `C++` Programmes. Dazu werden eingebaute Datentypen und auch einen Datentyp der Standardbibliothek vorgestellt. Auch die wichtigsten Kontrollstrukturen werden beschrieben und die erste eigene Funktion wird entwickelt.

Kapitel 3 Hier werden die die eingebauten Datentypen im Detail besprochen, sodass diese im echten Einsatz verwendet werden können.

Wie kann ich das Buch durcharbeiten?

Dieses Buch ist in einzelne Kapitel gegliedert, die so angelegt sind, dass man diese hintereinander, möglichst vom Anfang bis zum Ende durcharbeitet. Durcharbeitet deshalb, da dieses Buch viele Beispiele enthält, die zum Verständnis beitragen sollen. Probiere die Beispiele aus und teste die Programme! Das ist wichtig.

Es ist leider so, dass wenige kurze Teile dieses Buches so geschrieben haben werden müssen, dass sie sich auf Inhalte beziehen, die erst ein nachfolgenden Teile beschrieben sind. Dies habe ich dann jeweils auch vermerkt. Das bedeutet, dass diese Teile beim ersten Durcharbeiten nicht (vollständig) verstanden werden müssen. Diese Teile sind so verfasst, dass beim Durcharbeiten der nachfolgenden Abschnitte darauf zurückgegriffen werden kann.

Da ein Ziel dieses Buches ist, eine gut lesbare Anleitung zu sein, wird auf Wiederholungen des Stoffes weitgehend verzichtet. Wenn eine Information in einem Kapitel schon dargelegt worden ist, dann wird diese in den weiteren Kapitel als bekannt vorausgesetzt. Damit kann man nicht einfach ein Kapitel für sich alleine durcharbeiten, wenn man nicht über die Vorinformationen verfügt.

Es geht um modernes `C++`! Daher werde ich nicht explizit darauf hinweisen, dass ein Feature unter Umständen erst in `C++11` hinzugekommen ist. Wir gehen davon aus, dass wir einen `C++11` kompatiblen Compiler haben. Manche Features, die in `C++14` hinzugekommen sind, erwähne und erkläre ich auch. Diese sind dann allerdings klar gekennzeichnet, da die Untertützung durch die Compiler unter Umständen noch nicht gegeben ist.

Als Referenz ist das Buch [Stroustrup, 2013] vom Erfinder von `C++` zu empfehlen. Es behandelt weitgehend alle Aspekte der Sprache.

Benötigt man eine schnelle Referenz, dann rate ich dringend die Online-Hilfe unter [Kanapickas, 2014] aufzusuchen. Diese enthält wichtige Fakten zur Sprache und eine vollständige Referenz zur Standardbibliothek von `C++`!

Beispielprogramme

Jedes Thema wird an Hand von Beispielprogrammen demonstriert. Der praktische Ansatz steht im Vordergrund.

Für jedes Beispielprogramm wird ein Dateiname im Text (in der Kodierung UTF-8) angegeben, also könnte eine Datei zum Beispiel `hallo.cpp` heißen.

Diese Beispielprogramme werden in kleinen Schritten entwickelt, wobei für jeden relevanten Schritt der Sourcecode in einer eigenen Datei (mit fortlaufenden Nummer) auf der Website XXX zur Verfügung steht. Wenn das Programm `hallo.cpp` heißt, dann gibt es unter Umständen weitere Programme, die `hallo2.cpp`, `hallo3.cpp`,... heißen können.

Diese Beispielprogramme stehen als Sourcecode unter http://xxx.com zur Verfügung und sind je Kapitel zusammengefasst. Sie sollen eine Unterstützung bieten,

wenn einmal etwas nicht so funktioniert, wie man es sich erwartet, weil sich Fehler eingeschlichen haben.

Die Ausgaben meiner Programme beziehen sich auf ein 32 Bit Linux System, aber die Programme funktionieren auch unter 64 Bit Systemen als auch unter Windows und Mac OSX.

Feedback

Ich freue mich über jedes Feedback zu diesem Buch, das ich mit sehr viel Engagement und sehr sorgfältig ausgearbeitet habe. Leider gilt für ein Buch genau die gleiche Regel wie für jedes hinreichend großes Programm: Es gibt immer noch einen Fehler!

Wenn Sie in diesem Buch noch Fehler finden, eine Verbesserungsmöglichkeit sehen, einen Hinweis für Erweiterungen geben können oder einfach nur Bemerkungen für mich parat haben, dann bitte ich Sie diese mir per E-Mail an guenter.kolousek@gmail.com zu senden.

Anrede und Geschlechtsneutralität

Prinzipiell verwende ich in diesem Buch von nun an das informelle „Du“ anstatt dem formelleren „Sie“. Dies hat nichts mit mangelndem Respekt zu tun, den ich dir als Leser entgegenbringe, sondern trägt dem Umstand Rechnung, dass wir Programmierer doch eine Gemeinschaft sind und in dieser Gemeinschaft ist das deutsche „Du“ für die Kommunikation üblich.

In diesem Buch wende ich mich an Personen beiderlei Geschlechts, möchte jedoch nicht in eine der üblichen „geschlechtsneutralen“ Schreibweisen verfallen. Der folgende Text gibt ein Beispiel für solch eine Schreibweise an:

> Als Entwickler bzw. Entwicklerin bekommst du vom Auftraggeber bzw. der Auftraggeberin oft nur unklare Angaben bezüglich der Angaben zu den Kunden bzw. den Kundinnen und dessen bzw. deren Bedürfnissen.

Mir ist bewusst, dass es natürlich auch andere geschlechtsneutrale Schreibweisen verwendet werden, die meiner Meinung nach auch nicht sonderlich lesbarer sind bzw. der deutschen Sprache widersprechen.

Schauen wir uns einerseits das Wort „Lehrling“ (das österreichische Wort für Azubi) an, das grammatikalisch gesehen männlich ist und andererseits das Wort

„Koryphäe“, das in der deutschen Sprache weiblich ist. D.h. es heißt „der Lehrling“ und „die Koryphäe“ obwohl es natürlich Personen beiderlei Geschlechts gibt auf die diese Begriffe zutreffen.

In diesem Sinne verwende verwende ich, dem einfacheren Lesefluss zuliebe, die Form „der Programmierer“ und bitte die Leserinnen dieses Buches aus den angegebenen Gründen sich ebenfalls angesprochen zu fühlen.

Wer hat maßgeblich an der Entstehung dieses Buches beigetragen?

Hauptsächlich gebührt mein Dank meiner Frau Michaela, dich mich schon über so viele Jahre in meinem gesamten Werdegang unterstützt hat. Auch in diesem konkreten Fall des Schreibens an diesem Buch hat sie mich wieder einmal mit Tat und moralischer Unterstützung zum Gelingen dieses vorliegenden Werkes beigetragen. Dafür und auch für das nicht unwichtige Korrekturlesen möchte ich mich bei ihr ganz herzlich bedanken.

Weiters möchte ich mich bei (XXX Schüler) bedanken...

1 Einführung in C++

1.1 Charakterisierung von C++

Bei C++ handelt es sich um eine objektorientierte Programmiersprache, die eine der weltweit am häufigsten eingesetzten Programmiersprachen ist.

Die Entwicklung der Programmiersprache C++ wurde von Bjarne Stroustrup 1979 als eine objektorientierte Erweiterung zur Programmiersprache C unter dem Namen „C with Classes“ begonnen. 1984 erfolgte die Umbenennung in C++ und der erste kommerzielle Compiler erschien 1985. Seitdem entwickelt sich C++ permanent weiter und es folgten im Jahr 1989 und 2003 die Versionen C++98 und C++03. Diese Entwicklung hatte im Jahr 2011 mit C++11 einen vorläufigen Höhepunkt mit vielen substanziellen Neuerungen. C++11 legte den Grundstein für die moderne Entwicklung mit C++!

2014 wurde eine Zwischenversion mit Berichtigungen und kleinen Erweiterungen herausgebracht, die als C++14 bekannt ist. Man sieht, dass C++ sowohl eine lange Vergangenheit hat und es permanent weiter entwickelt wird: 2017 soll die nächste große Version von C++ erscheinen.

In diesem Sinne spreche ich in diesem Buch von „C++“ wenn ich eine beliebige Version von C++ meine, von „C++11“ wenn ich mich auf C++11 oder C++14 beziehe und von „C++14“ wenn ich speziell C++14 anspreche.

Der Erfinder von C++, Bjarne Stroustrup, beschreibt C++ als eine allgemein verwendbare Programmiersprache zum Entwickeln und Verwenden von eleganten und effizienten Abstraktionen.

Bjarne Stroustrup hat zwei grundlegende Entwurfsprinzipien für C++ festgelegt: Einerseits soll es möglich sein, mit C++ nahe der Hardware zu programmieren und andererseits sollen C++ Programme hoch performant sein.

Aus diesen beiden Vorgaben ergibt sich, dass C++ eine überragende Bedeutung im Bereich der Systemprogrammierung hat. Mit C++ werden Treiberprogramme, ganze Betriebssysteme, Compiler, Software für eingebettete Systeme (engl. embedded systems) wie Software in Autos, Industrieanlagen, Handys,... entwickelt.

C++ wird nicht nur zur Systemprogrammierung, sondern auch großteils zur Anwendungsprogrammierung verwendet. Die Arten dieser Anwendungen sind sehr vielfältig. Es werden Systeme für die Finanzmärkte, Simulationen, Büroanwendungen, Grafikprogramme, Spiele,... entwickelt. Große Bedeutung hat C++ auch im Grafikbereich, in der Verarbeitung von Multimediadaten wie Bildern, Video und Audioinformationen, bei der Visualisierung, in der Meteorologie, Physik, Genetik,...

Auch die Größe der Computer ist unterschiedlich auf denen diese Programmiersprache zum Einsatz kommt. Abgesehen von den eingebetteten Systemen, findet sich C++ im Einsatz auf Personal Computern, in Serverlösungen bis hin zu Supercomputer.

Große Firmen wie Adobe Systems Inc. (z.B. Photoshop), Apple Inc. (z.B. Teile von Mac OSX), Google Inc. (z.B. Google Chrome), Microsoft Corporation (z.B. Teile von Microsoft Windows), Mozilla Corporation (z.B. Firefox), Oracle Corporation (z.B. MySQL) und viele mehr setzen C++ in ihren Produkten ein. Bei den hier genannten Firmen handelt es sich um eine subjektive Auswahl ausgewählt nach bestem Wissen und Gewissen.

C++ wird allerdings von keiner Firma kontrolliert, sondern von einem internationalen Komitee entwickelt. Die einzelnen Versionen von C++ werden als Standard bei der ISO (International Organization for Standardization) herausgebracht und haben normativen Charakter.

Mit der Version von C++ aus dem Jahre 2011, also C++11, ist ein weiterer wichtiger Aspekt hinzugekommen: C++ soll leicht erlernbar sein. Um mit *allen* Aspekten von C++ vertraut zu sein, benötigt man viele Jahre intensives Auseinandersetzen und Übung. Um produktiv Programme in C++11 zu entwickeln, benötigt man nur *wenige Monate*.

Und genau das ist das Ziel dieses Buches: C++ in dem Maße zu erlernen, dass es möglichst schnell erlernt und produktiv verwendet werden kann!

1.2 Merkmale von C++

Typisierung

Je nach Art wie Variablen einem Typ zugeordnet wird, unterscheidet man verschiedene Arten der Typisierung. C++ wird als *statisch* und *streng* getypte Programmiersprache kategorisiert.

Statische Typisierung bedeutet, dass der Typ einer Variable schon bei Kompilierung feststeht. Kompilieren bedeutet, dass der Quelltext in eine ausführbare Datei übersetzt wird. Bei dynamisch getypten Programmiersprachen ist solch eine Kompilierung in der Regel nicht notwendig, da der Typ einer Variable erst zur Laufzeit feststeht.

Strenge Typisierung bedeutet hingegen, dass Operationen nur auf kompatiblen Typen durchgeführt werden und Typkonversionen nur beschränkt implizit durchgeführt werden. Das Gegenstück ist die schwache Typisierung, die Operationen weitgehend zwischen beliebigen Typen ermöglicht.

Es gibt keine exakte Definition von strenger oder schwacher Typisierung.

Unterstützte Programmierparadigmen

C++ ist eine Programmiersprache, die mehrere Programmierparadigmen unterstützt:

Imperative und prozedurale Programmierung Da du ja schon Programmierer in einer objektorientierten Programmiersprache bist, bist du prinzipiell auch in der Lage imperativ (basierend auf Befehlen und Anweisungen) und prozedural (strukturiert mittels Prozeduren und Funktionen) zu programmieren.

Generische Programmierung Mittels der generischen Programmierung versucht man Algorithmen möglichst allgemein zu definieren, sodass diese später mittels verschiedensten Datentypen verwendet werden können. Der Vorteil, der gegenüber einer überhandnehmenden Objektorientierung gesehen wird, ist, dass es in vielen Fällen eine einfachere und performantere Implementierung in statisch getypten Programmiersprachen bietet.

C++ bietet hierfür mittels der Templates einen ausgefeilten Mechanismus an, der generische Programmierung ermöglicht.

Objektorientierte Programmierung Die objekt-orientierte Programmierung ist eines der wichtigsten Paradigmen und bietet richtig angewandt enorme Vorteile bei der Abbildung des Problemraumes in ein adäquates Modell.

In C++ nimmt die objektorientierte Programmierung nicht mehr einen so großen Stellenwert ein wie früher. Das liegt daran, dass die generische Programmierung einen großen Anteil übernommen hat.

Funktionale Programmierung In der funktionalen Programmierung bestehen Programme ausschließlich aus Funktionen, deren Verarbeitung wie die

Berechnung mathematischer Funktionen gehandhabt wird. Diese Funktionen haben keinen Zustand und auch keine Nebeneffekte. Das kann Vorteile bringen und kann unter C++ auch in gewissen Maßen umgesetzt und genutzt werden.

Auch wenn C++ keine rein funktionale Sprache ist, bietet es doch mittels den Sprachkonstrukten Funktionen, Funktionsobjekten, Lambda-Ausdrücken und der Standardbibliothek (`std::function`, `std::bind`, Higher-order Funktionen in den Algorithmen, `std::async`) Möglichkeiten, um funktional zu programmieren.

Im weiteren Verlauf dieses Buches werden wir diese verschiedenen Seiten von C++ noch kennenlernen.

1.3 Benötigte Software

In diesem Abschnitt werde ich die *prinzipiellen* Möglichkeiten beschreiben, mit der C++ Programme erstellt werden können.

Es gibt mehrere Möglichkeiten, wie du deine Werkzeuge auswählen kannst.

Entwicklungswerkzeuge

Je nach Betriebssystem gibt es für die Entwicklungswerkzeuge wie Compiler, Linker und Debugger verschiedene Möglichkeiten, die im Abschnitt **A.1** auf der Seite **265** prinzipiell besprochen werden. Es liegt an dir eine der Varianten zu wählen.

Entwicklungsumgebung

Zum Erstellen und Verwalten des C++ Sourcecodes wird entweder eine integrierte Entwicklungsumgebung (IDE, engl. integrated develeopment environment) oder ein vernünftiger Editor benötigt.

Auch hier gibt es je nach verwendetem Betriebssystem mehrere Wahlmöglichkeiten. Es gibt allerdings auch die ausgezeichnete Entwicklungsumgebung QtCreator, die plattformübergreifend zur Verfügung steht.

Framework

Um Programme mit grafischer Oberfläche, mit Netzwerkfunktionalitäten oder mit Datenbankzugriffen zu entwickeln, wird noch ein Framework benötigt.

Bezüglich des verwendeten Frameworks ist die Zielplattform von entscheidender Bedeutung. Will man seine Programme exakt für Windows auf den Markt bringen, dann wählt man sinnvollerweise die entsprechende Möglichkeit von Microsoft. Ähnlich sieht die Situation für die Plattform Mac OSX und Linux aus.

Ist es allerdings das Ziel seine Programme sowohl unter Windows als auch unter Mac OSX und Linux zur Verfügung stellen zu wollen, dann bietet sich das plattformübergreifende Framework Qt an. Dieses Framework bietet die folgenden Vorteile:

- Da es plattformübergreifend ist, steht es für die Betriebssysteme Windows, Mac OSX und Linux zur Verfügung. Außerdem kann man damit auch Programme für die Betriebssysteme Android und iOS entwickeln!
- Es bietet uns die Möglichkeit portablen Code für grafische Oberflächen, Netzwerkfunktionen und Datenbankzugriffen produktiv zu entwickeln.
- Mit Qt wird außerdem die Entwicklungsumgebung QtCreator mitgeliefert, die für die Entwicklung von C++ Programmen ausgezeichnet geeignet ist.
- Für die Entwicklung von privaten Programmen oder von Open Source Programmen ist die Verwendung von Qt kostenlos.

Bei diesen überwältigenden Vorteilen stellt sich natürlich auch die Frage nach etwaigen Nachteilen:

- Unter Umständen ist dieses Framework nicht für *jeden* Anwendungsfall und für *jeden* Entwickler die optimale Lösung.

- Für die Entwicklung von kommerzieller Software muss eine Lizenz gekauft werden, wogegen die Entwicklung von privaten Programmen und Open Source Programmen kostenlos möglich ist.

  Bemerkung

  Eigentlich ist es so, dass sogar die Entwicklung von proprietärer Software unter Einhaltung der LGPL (Lesser General Public License) möglich ist. Dies kann dadurch erreicht werden, dass das infrage kommende Programm dynamisch gegen die Qt Bibliotheken gelinkt wird.

Das SDK (software development kit) von Qt kann von http://qt-project.org heruntergeladen und installiert werden.

Da du ein Programmierer bist, wird in weiteren davon ausgegangen, dass du dir deine Entwicklungsumgebung selbständig aussuchst und installierst.

1.4 Ausführbares C++

C++ ist eine Programmiersprache bei der Quelltext zuerst in die Maschinensprache der jeweiligen Zielplattform übersetzt werden muss. Das wird mittels eines Übersetzungsvorganges (auch Kompilierung genannt) von einem Compiler (eingedeutscht von engl. *compiler*) durchgeführt, der den Quelltext (engl. *source code*) in eine ausführbare Form (engl. *executable*) übersetzt.

Besteht das Programm aus mehreren Programmdateien, die getrennt übersetzt worden sind, dann müssen diese noch miteinander verbunden werden. Dieses Binden wird durch einen Linker (eingedeutscht von engl. *linker* oder *link editor*) durchgeführt. In der Regel muss man sich darum nicht kümmern, da dies durch das Compiler-Frontend, also das Compiler-Programm selbsttätig vorgenommen wird.

Die gängigen Implementierungen von C++ erwarten sich, dass der Quelltext in Dateien abgespeichert ist. Als Dateierweiterung (engl. *filename extension*) wird in C++ meist `.cpp` verwendet, aber auch `.C`, `.cc`, `.cxx` oder `.c++` kommen zum Einsatz. Wir verwenden `.cpp`.

Weiters gibt es sogenannte Headerdateien (wird später noch erklärt), deren Dateierweiterung meist `.h` besitzt, aber auch `.H` oder `.hpp` kommen zeitweise zum Einsatz. Wir verwenden `.h`.

Im Abschnitt **A.2** auf Seite **266** findest du eine Kurzanleitung wie aus einem C++ Quelltext mittels einer Shell eine ausführbare Datei erstellt werden kann.

Danach kann das ausführbare Programm gestartet werden, es entsteht ein Prozess, der sich irgendwann einmal beenden wird. Beim Beenden gibt der Prozess einen Exitcode an den Prozess zurück, der ihn gestartet hat. Mittels dieses Exitcodes kann der beendete Prozess eine Information an den aufrufenden Prozess zurückgeben.

Im Anhang **A.3** auf der Seite **269** befindet sich eine Kurzanleitung wie du je, verwendetem Betriebssystem, auf diesen Exitcode mittels der Shell zugreifen kannst.

2 Einstieg in C++

Dieses Kapitel gibt einen Überblick über die prozedurale Programmierung mit C++ und führt die wichtigsten syntaktischen Elemente ein. Am Ende bist du in der Lage einfache textorientierte Programme zu schreiben.

Folgende Konzepte werden *überblicksmäßig* beschrieben:

- Programmgerüst und Kommentare
- Einfache Ausdrücke und Anweisungen
- Variable und Datentypen: C-Strings und `string`, Zeichen, Zahlen, Wahrheitswerte
- Präprozessoranweisungen, Namensräume
- Verarbeitung der Kommandozeilenargumente
- Ein- und Ausgabe über die Standardkanäle
- Zusammengesetzte Anweisungen: Bedingte Anweisung, Auswahl- und Schleifenanweisungen
- Arrays und Pointer
- Klasse `vector` und Verwendung von `sort`
- Definition einer eigenen Funktion

2.1 Das minimale Programm

Jedes C++ Programm muss *genau* eine Funktion haben, die `main` heißt. Diese Funktion `main` wird ausgeführt, wenn das Programm gestartet wird. Für das kürzeste C++ Programm sieht diese Funktion `main` aus wie im nachfolgenden Quelltext zu ersehen:

```
1 int main() {
2 }
```

Schreibe diesen Text in eine Datei mit dem Namen `minimal.cpp` und übersetze diese Datei in ein ausführbares Programm `minimal` bzw. `minimal.exe` unter Windows. Führe danach dieses Programm folgendermaßen aus:

```
1  $ minimal
```

Das Zeichen `$` gefolgt von einem Leerzeichen sind meine definierten Promptzeichen, die bei dir durchaus anders aussehen können. Der Prompt ist das Zeichen deiner Shell, dass sie auf eine Eingabe wartet. Oft wird im Prompt auch noch das aktuelle Verzeichnis oder zusätzliche Information angezeigt. Bei mir sind es lediglich die beiden Zeichen, damit du siehst was eingegeben wird und was die Ausgabe ist.

Du bemerkst, dass es zu keiner Ausgabe kommt. Das ist natürlich klar, da keinerlei Anweisungen im Programm enthalten sind.

Eine Funktion ist in C++ so aufgebaut, dass zuerst der Typ des Rückgabewertes angegeben wird, danach der Name der Funktion, gefolgt von der Parameterliste in runden Klammern und am Ende folgt der Rumpf der Funktion in geschwungenen Klammern.

Wir bezeichnen in weiterer Folge einen Abschnitt, der beliebig viele Anweisungen (engl. *statements*) enthalten kann und mit einer offenen und einer schließenden geschwungenen Klammer eingeschränkt ist, als einen Block (engl. *block*).

Damit stellt der Funktionsrumpf ebenfalls einen Block dar.

Der Typ des Rückgabewertes der Funktion `main` muss `int` sein. `int` ist eine Typangabe für ganze Zahlen (engl. *integer*) in C++. Es ist für die Funktion `main` allerdings nicht erforderlich, dass sie tatsächlich einen Wert zurückliefert. Wir kommen etwas später darauf zu sprechen.

Bei `int` handelt es sich weiter um ein Schlüsselwort (engl. *keyword*, eingedeutscht Keyword). Ein Keyword ist Teil der Programmiersprache und kann nicht als Name einer Funktion, einer Variable,... verwendet werden.

Die Parameterliste ist in dieser von uns derzeit verwendeten Form für die Funktion `main` leer und auch der Rumpf dieser Funktion ist derzeit noch leer. Damit wird dieses Programm bei der Ausführung auch überhaupt nichts bewirken.

Ein Programm besteht aus einer Folge von Anweisungen. Anweisungen werden in C++ herangezogen, um die Reihenfolge der Ausführung der einzelnen Prozessorinstruktionen anzugeben.

Viele Anweisungen werden in C++ oft mit einem Strichpunkt (engl. *semicolon*) abgeschlossen. Die leere Anweisung ist einfach ein einzelner Strichpunkt. Das folgende Programm besitzt ebenfalls keine Funktionalität, da im Rumpf der Funktion `main` nur eine leere Anweisung steht:

```
1  int main() {
2      ;
3  }
```

Es ist zu erwähnen, dass in C++ sogenannte Whitespace-Zeichen wie einzelne Leerzeichen, Leerzeilen und Tabulatorzeichen ignoriert werden. Davon ausgenommen sind natürlich Whitespace-Zeichen in Strings.

Damit im Rumpf der Funktion `main` irgendetwas steht, schreiben wir zwei Kommentare hinein:

```
1  int main() {
2      // das ist ein einzeiliger Kommentar
3      /* und dieser Kommentar erstreckt sich
4         über mehrere Zeilen.
5      */
6  }
```

Einzeilige Kommentare werden mit den Zeichen `//` eingeleitet und enden mit dem Ende der Zeile. Jedes beliebige Zeichen kann in einen einzeiligen Kommentar geschrieben werden.

Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/` und können mit Ausnahme von `*/` jede beliebige Zeichenkombination enthalten. `*/` darf nicht enthalten sein, da diese Zeichenkombination den mehrzeiligen Kommentar beendet. Damit sind keine verschachtelten mehrzeiligen Kommentare möglich.

2.2 Rückgabewert der Funktion `main`

Jetzt erinnern wir uns daran, dass die Funktion `main` einen Rückgabewert `int` haben muss und dass dieser bei `main` nicht angegeben werden muss.

Damit stellen sich drei Fragen:

- Wie kann man auf den Rückgabewert zugreifen?

  Es hängt davon ab, wie das Programm, das den Rückgabewert zurückliefert gestartet worden ist. Das kann einerseits durch ein „normales“ Programm (z.B. auch in C++ geschrieben) oder durch eine Shell erfolgen.

  Für unser Ausprobieren ist es am einfachsten den Rückgabewert nach dem Ausführen unseres Programmes durch die Shell abzufragen. Dieses Zugreifen auf den Rückgabewert erfolgt je Betriebssystem und Shell unterschiedlich. Eine Kurzanleitung ist im Anhang im Abschnitt **A.3** auf der Seite **269** zu finden.

- Wie kann eine Funktion einen Rückgabewert zurückgeben?

  Eine Funktion kann einen Wert mittels des Schlüsselwortes `return` zurückgeben:

```
1  int main() {
2      return 0;
3  }
```

  In diesem Fall wird 0 zurückgegeben. Das bedeutet, dass der Exit-Code in der Shell dann ebenfalls 0 beträgt. Der Wert 0 wird in der Regel als Erfolg gedeutet, während allen anderen Werten je Anwendung unterschiedliche Bedeutung zugewiesen wird.

- Welchen Wert erhält man, wenn man keinen Wert von `main` zurückgibt?

  Durch Ausprobieren kannst du leicht herausfinden, dass defaultmäßig 0 zurückgeliefert wird. `main` ist die einzige Funktion, die in C++ explizit keinen Wert zurückliefern muss (mittels `return`), obwohl sie einen Typ als Rückgabewert (nämlich `int`) angegeben hat.

2.3 Hello World

Jetzt nehmen wir die nächste Hürde und schreiben das klassische „Hello World“ Programm in eine Datei `hello.cpp`:

```
1  // hello.cpp
2  #include <iostream>
3
4  int main() {
5      std::cout << "Hello, world!" << std::endl;
6  }
```

Nach dem Übersetzen und Linken kommt es zu folgender erwarteter Ausgabe bei dem Ausführen des Programmes:

```
1  $ hello
2  Hello, world!
```

Ok, und was passiert eigentlich im Programm?

- In Zeile 2 wird eine sogenannte Header-Datei eingebunden. Das bedeutet, dass der Inhalt der angegebenen Datei `iostream` an dieser Stelle, an der die `#include` Direktive (Befehl) steht eingelesen wird. Damit steht alles was sich in dieser Datei befindet an dieser Stelle zu Verfügung. Dieses Einbinden wird *vor* dem eigentlichen Übersetzen durch einen Präprozessor (engl. *preprocessor*) durchgeführt. Der Compiler sieht danach sowohl den Inhalt der eingebundenen Datei als auch den Rest der Datei `hello.cpp`.

  Der Präprozessor wird in C-basierten Sprachen dazu verwendet, Textersetzungen vorzunehmen, bevor der eigentliche Compiler den Quellcode bekommt. Der Präprozessor liest die Präprozessordirektiven, die immer mit dem Rautezeichen `#` (Hashzeichen, Nummernzeichen) beginnen und führt diese aus.

  In Header-Dateien sind Deklarationen enthalten. Eine Deklaration ist zum Beispiel eine Funktionsdeklaration, die den Namen, den Rückgabewert und die Typen der Parameter festlegt. Ein Name einer Funktion (oder Variable, Typ,...) wird als Bezeichner (engl. *identifier*) bezeichnet.

  Im Fall der Header-Datei `iostream` sind unter anderem die Deklarationen von `cout` und `endl` enthalten. Allerdings sind diese in einem Namensraum (engl. *namespace*) `std` enthalten auf die hier mittels `::` zugegriffen wird.

  Ein Namensraum ist ein Behälter, der Bezeichner beinhalten kann, die unabhängig von Bezeichnern außerhalb dieses Behälters sind. Ein Beispiel aus dem

täglichen Leben wären die Vorwahlen von Telefonnummern. Eine Vorwahl dient als Behälter für die nachfolgende Rufnummer, sodass es zu keinem Konflikt mit einer gleichen Rufnummer aber einer anderen Vorwahl kommt.

Als einfache Regel gilt: Alle Bezeichner der Standardbibliothek von C++ sind im Namensraum `std` enthalten.

- In der Zeile 5 wird auf das Objekt `cout` im Namensraum `std` mittels des Operators `::` (Bereichsauflösungsoperator, engl. *scope resolution operator*) zugegriffen.
- In Zeile 5 wird weiters mittels des Operators `<<` die C-String-Konstante `"Hello, world!"` (engl. *C string literal*) ausgegeben. Werte, die direkt im Quellcode geschrieben werden können, bezeichnet man als Literal (engl. *literal*). Beispiele sind eben hier das C-String-Literal `"Hello, world!"` oder die ganze Zahl `3`.

  Bei `std::cout` handelt es sich um ein Objekt, das den `stdout` Kanal repräsentiert. Alle Daten, die in diesen Kanal geschrieben werden, werden auf der Shell zur Ausgabe gebracht, wenn das Programm mittels einer Shell gestartet worden ist.

  Der Operator `::` liefert wieder das ursprüngliche Objekt `cout` zurück, sodass wiederum der Operator `<<` verwendet werden kann. Dies wird ausgenützt, um einen Zeilenumbruch zu erzwingen, da danach der Manipulator `std::endl` (*end line*, Zeilenende) an die Ausgabe gesendet wird.
- Die Zeile 5 beinhaltet genau eine einfache Anweisung, die mit einem Strichpunkt (engl. *semicolon*) abgeschlossen sein muss.

Weiter oben wurde noch von dem „Linken“ gesprochen, das in unserem konkreten Fall `std::cout` und `std::endl` sowie die Operatorfunktion(en) `<<` mit den entsprechenden Definitionen verbinden muss (siehe Abschnitt **3.4** auf Seite **60**). Normalerweise wird die Funktion des Präprozessors, das eigentliche Übersetzen und auch das Linken von einem Programm übernommen.

2.4 Eingabe und Ausgabe

Bis jetzt haben wir lediglich Ausgaben getätigt, nun werden wir uns auch mit der Eingabe beschäftigen.

Um Daten einzugeben, existiert das Objekt `std::cin`, das den Kanal `stdin` repräsentiert und die Eingabe zur Laufzeit des Programmes ermöglicht. Dafür

wird ebenfalls die Headerdatei `iostram` benötigt und die Funktionsweise von `std::cin` ist analog zu `std::cout`. Schreiben wir gleich einmal das nächste Programm, das eine Abwandlung unseres „Hello World“ Programmes darstellt. Es soll den Benutzer nach seinem Namen fragen und ihn danach nett begrüßen:

```
1  // greetme.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      string name;
8      cout << "Ihr Name bitte: ";
9      cin >> name;
10     cout << "Nett Sie zu sehen, " << name << "!" << endl;
11 }
```

Übersetze das Programm wie gewohnt und starte es, du wirst nach deinem Namen gefragt und danach wird der Begrüßungstext ausgegeben:

```
1  $ greetme
2  Ihr Name bitte: Maxima
3  Nett Sie zu sehen, Maxima!
4  $
```

Gut was ist hier Neues zu finden?

- Die Zeile 4 legt fest, dass alle Bezeichner aus dem Namensraum `std` ab jetzt nicht mehr mit dem Bereichsauflösungsoperator `::` qualifiziert werden müssen, sondern einfach verwendet werden können. Das heißt also, dass wir den Namensraum `std` von nun an benutzen und dies gilt bis zum Ende der Datei.

- In Zeile 7 wird hier zum ersten Mal eine Variable definiert. In Zeile 6 wird `name` als eine neue Variable definiert, die den Typ `string` hat. `string` ist ein C++ Typ, der eine Zeichenkette repräsentiert, aber im Gegensatz zu den C-Strings ein Objekttyp ist.

In der folgenden Zeile `cout << "Ihr Name bitte: ";` siehst du wieder eine C-String-Konstante. Bei einem C-String handelt es sich lediglich um eine Folge von Zeichen im Speicher, die mit einem Nullzeichen (`'\0'`, ein Wert mit allen Bits 0) abgeschlossen ist und durch eine Adresse im Speicher referenziert wird. Eine C-String-Konstante ist eine Möglichkeit, wie ein C-String im Quelltext angegeben werden kann.

Da es sich bei `string` um einen Objekttyp handelt, gibt es dafür auch eine Klasse. Die Deklaration dieser Klasse steht in der Headerdatei `string`, die wiederum mit `#include <string>` eingebunden werden hätte müssen. Allerdings bindet die Headerdatei `iostream` diese Headerdatei `string` von selber ein und deshalb kann es in diesem Fall unterlassen werden.

Wir werden uns den Objekttyp `string` später genauer ansehen und auch die Unterschiede zu einem C-String herausarbeiten.

- Weiters wird diese Variable `name` mittels des `>>` Operators aus dem Objekt `cin` befüllt. Da es sich bei dem Typ von `name` um einen `string` handelt, werden die eingegebenen Zeichen – so wie sie sind – in diese Variable gespeichert.

So weit, so gut. Vielleicht hast du es schon ausprobiert einen vollständigen Namen mit Vornamen und Nachnamen einzugeben, dann wirst du folgendes Verhalten festgestellt haben:

```
1 Ihr Name bitte: Maxima Muster
2 Nett Sie zu sehen, Maxima!
```

Was passiert hier? Der Operator `>>` liest maximal bis zum ersten Leerzeichen und damit erhalten wir in unserem Fall nur den Vornamen! Ändern wir daher das Programm nochmals ab, sodass der Vorname als auch der Nachname eingegeben werden kann:

```
1  // greetme2.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      string first_name;
8      string last_name;
9      cout << "Ihr Name bitte: ";
10     cin >> first_name;
11     cin >> last_name;
12     cout << "Nett Sie zu sehen, " << first_name
13          << " " << last_name << "!" << endl;
14 }
```

In diesem Programm lesen wir einfach zwei Strings ein und speichern diese in zwei Variablen ab, die danach mit einem Leerzeichen getrennt ausgegeben werden:

```
1  Ihr Name bitte: Max Muster
2  Nett Sie zu sehen, Max Muster!
```

Diese Lösung hat allerdings den Nachteil, dass nämlich jetzt wirklich zwei Namen eingegeben werden müssen. Probiere einfach einmal aus, nur den Vornamen einzugeben und du wirst bemerken, dass das Programm nach der ersten Eingabe noch auf eine zweite Eingabe wartet!

Was kann man bei der Ausführung dieses Programmes also tun?

- Einerseits kann natürlich das Programm abgebrochen werden. Das kann in der Regel mit der Tastenkombination CTRL+C erreicht werden. In diesem Fall bricht aber das Programm eben ab und es wird zu keiner Ausgabe kommen.
- Die zweite Möglichkeit besteht darin, dem Programm mitzuteilen, dass die Eingabe beendet ist, d.h. dass der Eingabekanal für diese Operation geschlossen wird. Das kann unter Linux oder Mac OSX mittels der Tastenkombination CTRL+D und unter Windows mittels CTRL+Z erreicht werden. D.h. nach der

erfolgreichen Eingabe des ersten Wertes wird die zweite Eingabe mit einem Leerstring versehen.

Bei beiden Varianten handelt es sich eigentlich um keine vernünftigen Lösungen.

Verändern wir deshalb das Programm nochmals:

```
1  // greetme3.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      string name;
8      cout << "Ihr Name bitte: ";
9      getline(cin, name);
10     cout << "Nett Sie zu sehen, " << name << "!" << endl;
11 }
```

In diesem Programm verwenden wir die Funktion `getline`, die sowohl das Objekt `cin` als auch eine Variable vom Typ `string` mitbekommt. Die Funktionsweise ist, dass eine gesamte Zeile des Eingabekanals in die Variable `name` eingelesen wird. Beachte, dass es in C++ möglich ist, Parameter zu übergeben, die in der Funktion verändert werden, wie dies bei `name` zu sehen ist und wie es die Funktion `getline` handhabt.

Weiters siehst du auch, dass diese Funktion `getline` anscheinend in der Headerdatei `iostream` enthalten ist, da der Compiler keinen Fehler meldet.

Durch diese Lösung ist natürlich auch keine Aufteilung in Vorname und Nachname mehr möglich, dafür sind aber auch mehrere Namensteile wie mehrere Vornamen oder Doppelnamen möglich. Probiere es einfach aus!

Es könnte sein, dass die Behandlung von Sonderzeichen unter Umständen nicht korrekt funktioniert. Das liegt daran, dass dein System wahrscheinlich nicht richtig konfiguriert ist. Kontrolliere deine Systemeinstellungen!

2.5 Ganze Zahlen und `if`

Im nächsten Schritt wollen wir uns mit einem weiteren grundlegenden Datentyp, nämlich den ganzen Zahlen beschäftigen und als erste Kontrollstruktur die `if` Anweisung einsetzen.

Schreiben wir jetzt ein Programm, das das Alter des Benutzers abfragt und danach eine Altersüberprüfung vornimmt. Ist das Alter unter 18 Jahren, dann soll ausgegeben werden, dass der Benutzer noch nicht volljährig ist, ansonsten, dass er volljährig ist.

Schreibe deshalb folgendes Programm:

```
1  // agetest.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int age;
8
9      cout << "Wie alt sind Sie? ";
10     cin >> age;
11
12     if (age < 18) {
13         cout << "Sie sind noch minderjährig!" << endl;
14     } else {
15         cout << "Sie sind volljährig!" << endl;
16     }
17 }
```

Probiere das Programm jetzt aus und dann werden wir es besprechen.

Fertig? Dann schauen wir uns das einmal an:

- Zuerst sehen wir, dass wir eine Variable `age` vom Typ `int` (Abkürzung für *integer*, ganze Zahl) definieren. Das heißt, wir drücken aus, dass wir eine Variable mit dem Bezeichner „age“ wollen, die ganze Zahlen speichern kann.
- Weiters sehen wir, dass wir den Operator `>>` nicht nur zur Eingabe von Strings, sondern auch zur Eingabe von ganzen Zahlen verwenden können. Der Benut-

zer gibt eine Zeichenkette ein und `>>` wandelt diese in eine ganze Zahl um, wenn dies möglich ist (mehr dazu später).

Weiters gehen wir davon aus, dass der Benutzer nur erlaubte Eingaben tätigt. Wir werden uns später noch eingehend damit auseinandersetzen, wie man mit fehlerhaften Eingaben umgehen kann.

- Außerdem sehen wir danach die erste zusammengesetzte Anweisung, nämlich die Bedingungsanweisung, die die Bedingung nach dem Schlüsselwort `if` (zu Deutsch: wenn, falls) in runden Klammern enthält. Ist die Bedingung wahr, dann werden die direkt folgenden Anweisungen des ersten Anweisungsblockes (in den geschwungen Klammern) ausgeführt, anderenfalls die Anweisung nach dem Schlüsselwort `else` (dt. anderenfalls).

  Bedingungen setzen sich meist aus Vergleichsoperatoren und logischen Operatoren zusammen.

  An Vergleichsoperatoren stehen uns für Zahlen die üblichen Kandidaten `<` (kleiner), `<=` (kleiner gleich), `!=` (ungleich), `==` (gleich), `>=` (größer gleich) und `>` (größer) zur Verfügung.

  Die logischen Operatoren werden wir später noch besprechen.

  Ein `else` Teil ist nicht notwendig. Dieser kann auch weggelassen werden. Damit kann das Programm auch ohne Verwendung von `else` Teilen umgeschrieben werden. Dies ist zwar nicht so elegant und lesbar, aber möglich. Die entsprechenden Bedingungsanweisungen sehen danach folgendermaßen aus:

```
1  if (age < 18) {
2      cout << "Sie sind noch minderjährig!" << endl;
3  }
4
5  if (age >= 18) {
6      cout << "Sie sind volljährig!" << endl;
7  }
```

- Die Anweisungen in einer zusammengesetzten Anweisung wie der `if` Anweisung müssen prinzipiell in geschwungenen Klammern geschrieben werden, wie wir es schon gesehen haben.

  Es gibt allerdings eine Ausnahme: Handelt es sich jeweils nur um eine einzelne Anweisung, dann können die geschwungenen Klammern entfallen, d.h. die

Verzweigungsanweisung kann in diesem Fall auch folgendermaßen geschrieben werden:

```
1  if (age < 18)
2      cout << "Sie sind noch minderjährig!" << endl;
3  else
4      cout << "Sie sind volljährig!" << endl;
```

Das bedeutet, dass Anweisungen innerhalb von geschwungenen Klammern überall dort verwendet werden können wo auch eine einzelne Anweisung steht.

Ich empfehle diese Vorgangsweise aber *nicht*: Fügst du später weitere Anweisungen ein, dann kann man leicht darauf vergessen die geschwungenen Klammern hinzuzufügen und das Programm lässt sich entweder nicht übersetzen oder es wird nicht richtig funktionieren.

Wie sieht es aus, wenn wir eine feinere Unterscheidung haben wollen:

Altersgruppe	von	bis
Kind	0	<14
Jugendlicher	14	<18
Erwachsener	18	-

Mit den uns bekannten Mitteln kann man diese Aufgabenstellung ohne Probleme folgendermaßen lösen. Beachte, dass ich jetzt absichtlich keine geschwungenen Klammern gesetzt habe:

```
1  if (age < 18)
2      if (age < 14)
3          cout << "Du bist noch ein Kind." << endl;
4      else
5          cout << "Sie sind ein Jugendlicher!" << endl;
6  else
7      cout << "Sie sind volljährig!" << endl;
```

Das ist soweit verständlich und funktionsfähig. Nehmen wir nun einmal an, dass uns nicht interessiert, ob jemand jugendlich ist oder nicht. Uns interessiert nur, Kind oder Erwachsener. Die naheliegende Lösung ist, den entsprechenden `else` Zweig einfach zu löschen:

```
1  if (age < 18)
2      if (age < 14)
3          cout << "Du bist noch ein Kind." << endl;
4  else
5      cout << "Sie sind volljährig!" << endl;
```

Damit funktioniert das Programm nicht mehr gemäß unseren Erwartungen. Wir haben zwar sinnvoll eingerückt, aber in C++ wird Whitespace nicht betrachtet. Damit zählt das `else` zur zweiten `if` Anweisung! Das wäre nicht passiert, hätten wird geschwungene Klammern verwendet!

Allerdings sollte man solche Art von verschachtelten Verzweigungen eher vermeiden. Schreibt man das Programm um, dann funktioniert es wie erwartet und ist auch besser zu lesen:

```
1  if (age < 14) {
2      cout << "Du bist noch ein Kind." << endl;
3  } else if (age < 18) {
4      cout << "Sie sind ein Jugendlicher!" << endl;
5  } else {
6      cout << "Sie sind volljährig!" << endl;
7  }
```

2.6 Addieren von ganzen Zahlen

Nehmen wir einmal an, wir wollen ein einfaches Programm schreiben, das einfach nur zwei Zahlen einliest, die erste Zahl zu der zweiten Zahl addiert und das Ergebnis ausgibt.

Eine sehr einfache Problemstellung, aber trotzdem können wir hier wieder einiges über C++ lernen. Beginnen wir gleich einmal mit dem folgenden Programm:

```
1  // add.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int num1;
8      int num2;
9      int res;
10
11     cout << "Die erste Zahl: ";
12     cin >> num1;
13     cout << "Die zweite Zahl: ";
14     cin >> num2;
15
16     cout << "Das Ergebnis ist: " << res << endl;
17 }
```

Was gibt es hier wieder Neues? Eigentlich gar nichts. Aber starte einmal das Programm, gib beliebige Werte für die beiden Zahlen ein und schaue dir die Ausgabe an. Sie wird in *etwa* folgendermaßen aussehen und wird bei dir vermutlich ein anderes Ergebnis anzeigen.

```
1  $ div
2  Die erste Zahl: 1
3  Die zweite Zahl: 2
4  Das Ergebnis ist: -1218654208
```

Die Variable `res` wurde **nicht** initialisiert, d.h. es wird kein Wert in diese Variable geschrieben. Was ist aber der Wert von `res` nach der Definition? Die Variable enthält den Wert, den der Speicherinhalt zu dieser Zeit repräsentiert. Damit ist es vom aktuellen Bitmuster des Speicherinhaltes abhängig.

Die Variable `res` ist eine lokale Variable, da diese innerhalb der Funktion `main` definiert wurde. Merke: Lokale Variable, die einen fundamentalen Datentyp (eingebaute Datentypen wie ganze Zahlen, Gleitkommazahlen,...) besitzen werden in

C++ **nie** automatisch initialisiert. Wir werden später noch auf die fundamentalen Datentypen eingehen.

Was können wir also machen? Wir könnten nach der Definition der Variable `res` dieser einen Wert *zuweisen*:

```
1 int res;
2
3 res = 0;
```

Ok, das ändert aber nichts daran, dass diese Variable am Beginn noch immer nicht initialisiert wurde. Gut, dann lernen wir gleich wie man eine Variable in C++11 korrekt initialisiert:

```
1 int res{0};
```

D.h. der Initialisierungswert wird in geschwungenen Klammern nach dem Variablennamen geschrieben. In diesem konkreten Fall ist der Effekt gleich wie wenn man die Variable nicht initialisiert und ihr dann einen Wert zuweist. Verwendet man jedoch komplexe Objekte, dann kann dies zu einem Performanceverlust führen.

Faustregel

Lokale Variable **immer** initialisieren!

Es gibt eigentlich nur eine Ausnahme von dieser Regel: Hat das zu initialisierende Objekt einen hohen Initialisierungsaufwand und wird es zuerst beschrieben und erst dann gelesen, kann man die Initialisierung weglassen. Im Abschnitt **5.2** findest du auf der Seite **134** eine Beschreibung der Auswirkung der Initialisierung eines Arrays.

Allerdings geht es mit der vereinheitlichten Initialisierung in diesem Fall noch einfacher:

```
1 int res{};
```

Du siehst, dass du die `0` weglassen kannst. In diesem Fall wird die Variable mit dem „Nullwert“ initialisiert, falls es sich um einen sogenannten fundamentalen Datentyp handelt.

Es besteht auch die Möglichkeit den Wert direkt bei der Definition der Variable anstatt der geschwungenen Klammern mit einem `=` Zeichen zu initialisieren:

```
1  int res = 0;
```

Das hat den gleichen Effekt. Trotzdem empfehle ich die Form mit den geschwungenen Klammern, die als einheitliche Initialisierung (engl. *uniform initialization*) bezeichnet wird und in `C++11` zum Sprachstandard hinzugekommen ist. Der Vorteil dieser Form ist, dass diese weitgehend *überall* in `C++` zur Initialisierung verwendet werden kann (im Kontrast zu der Form mit `=`).

Ändern wir jetzt das Programm so ab, dass es die gewünschte Addition durchführt. Füge dazu direkt nach der Eingabe der zweiten Zahl folgende Anweisung in das Programm ein:

```
1  res = num1 + num2;
```

Damit wird die erste Zahl zur zweiten Zahl addiert und in das Ergebnis der Variable `res` zugewiesen. Schaut man sich das Programm jetzt an, dann sieht man unschwer, dass die Initialisierung der Variable `res` jetzt nicht mehr nötig wäre. Wir lassen das Programm trotzdem in dieser Form, da wir unsere Faustregel beachten!

Das Zuweisen des Ergebnisses zu der Variable `res` geschieht mittels des Zuweisungsoperators `=`. Es handelt sich hier **nicht** um eine Initialisierung, sondern um eine Zuweisung. Das ist wichtig für das weitere Verständnis von `C++`.

Als ein Ausdruck (engl. *expression*) wird ein syntaktisches Konstrukt bezeichnet, das zu einem *Wert* ausgewertet werden kann. `num1 + num2` ist ein Ausdruck. In `C++` ist `res = num1 + num2` ebenfalls ein Ausdruck, da der Wert dieses Ausdruckes das Ergebnis der Addition ist.

Dieser Ausdruck `res = num1 + num2` beinhaltet eben einen weiteren Ausdruck, nämlich `num1 + num2`, der wiederum zwei Ausdrücke beinhaltet, nämlich `num1` und `num2`. Wir sehen, dass ein Ausdruck wieder aus (Teil-)Ausdrücken bestehen

kann. Als vollständiger Ausdruck (engl. *full expression*) wird ein Ausdruck verstanden, der nicht Teil eines anderen Ausdruckes ist. In unserem konkreten Fall handelt es sich bei `res = num1 + num2` um einen vollständigen Ausdruck und bei `num1 + num2` beziehungsweise bei `num1` oder `num2` um *keine* vollständigen Ausdrücke.

Hängt man an einen Ausdruck einen Strichpunkt, wird daraus eine Anweisung! Damit handelt es sich bei `res = num1 + num2;` um eine Anweisung. Im Unterschied zu einem Ausdruck hat eine Anweisung *keinen* Wert.

Die anderen grundlegenden arithmetischen Operatoren sind `-` (Subtraktion), `*` (Multiplikation) und `/` (Division). `C++` kennt selbstverständlich die üblichen Regeln zur Punkt- und Strichrechnung, wobei Klammern gesetzt werden können, um eine beliebige Berechnungsreihenfolge herzustellen (z.B. `a * (b + c)`).

Jedem Operator ist nicht nur eine Priorität zugewiesen, die die Reihenfolge der Berechnung festlegt, sondern auch eine Assoziativität. Die Assoziativität bestimmt wie Ausdrücke wie `a + b + c` ausgerechnet werden. `a + b + c` wird von links nach rechts ausgerechnet (links assoziativ) und ist damit gleichwertig zu `(a + b) + c`. Damit muss man die Klammern bei diesem speziellen Ausdruck nicht verwenden, sie sind in diesem Fall redundant.

So, übersetze das Programm wieder und probiere es aus:

```
1 Die erste Zahl: 1
2 Die zweite Zahl: 2
3 Das Ergebnis ist: 3
```

Das Ergebnis erwartet man sich natürlich. Damit wäre die geforderte Funktionalität erfüllt.

Schauen wir uns trotzdem an was passiert, wenn der Benutzer ungültige Werte eingibt:

- Probiere folgende Interaktion aus:

```
1 Die erste Zahl: 1
2 Die zweite Zahl: 2a
3 Das Ergebnis ist: 3
```

Was passiert hier? Die erste Eingabe ist klar, es wird die ganze Zahl 1 eingelesen. Bei der zweiten Eingabe sieht es so aus, dass eine 2 eingelesen wird und das Zeichen `a` weiter im Eingabestrom gelassen wird.

Schauen wir uns das einmal an, indem wir den folgenden Text hinten als letzte Anweisung im Rumpf von `main` schreiben:

```
1  string rest;
2  cin >> rest;
3  cout << "Der verbleibende Rest: " << rest << endl;
```

`cin >> rest` liest genau ein Zeichen in die neue Variable `rest` ein und dieses wird danach ausgegeben.

Probiere jetzt dein Programm mit den Eingaben wieder aus. Du siehst, dass das Zeichen `a` als Rest ausgegeben wird. Du siehst, dass die zweite Zahl nur soweit eingelesen wird, bis eine vollständige Zahl erkannt wird. Hier folgt die Ausgabe:

```
1  Die erste Zahl: 1
2  Die zweite Zahl: 2a
3  Das Ergebnis ist: 3
4  Der verbleibende Rest: a
```

- Starte jetzt dieses Programm noch einmal und gib nur gültige Eingaben ein. Du bemerkst, dass die Anweisung `cin >> rest` sich jetzt die Eingabe eines Zeichens erwartet und du daher eines eingeben musst. Allerdings wird kein Leerzeichen akzeptiert, da dieses ja überlesen wird.

  Entweder du gibst ein beliebiges Zeichen ein oder du brichst die Eingabe ab. Das geht unter Windows mit der Tastenkombination `CTRL-Z` und unter Mac OSX oder Linux mittels `CTRL-D`.

- Starte jetzt das Programm noch einmal und gib zuerst `1a` ein und drücke wie gewohnt die Return-Taste. Die Ausgabe wird folgendermaßen aussehen:

```
1 Die erste Zahl: 1a
2 Die zweite Zahl: Das Ergebnis ist: 1
3 Der verbleibende Rest:
```

Tja, hier sehen wir, dass es überhaupt nicht funktioniert hat. Die erste Zahl wird richtig gelesen, die zweite Zahl kann nicht mehr gelesen werden, da lediglich das Zeichen `a` im Eingabestrom vorhanden ist. Damit handelt es sich aus Sicht von C++ um eine fehlerhafte Eingabe, da eine ganze Zahl erwartet wird. Das bedeutet, die fehlgeschlagene Eingabe bewirkt, dass `num2` auf den Nullwert gesetzt wird und der Eingabestrom in einen Fehlermodus versetzt wird. Damit schlägt auch die Eingabe des Rests fehl!

Wie man mit solchen Situationen umgeht, werden wir uns noch anschauen. Jetzt einmal ist es wichtig zu erkennen, wie in C++ die Eingabe prinzipiell funktioniert. Hier noch einmal eine Kurzzusammenfassung:

- Es werden Leerzeichen prinzipiell überlesen.
- Es werden dann alle Zeichen eingelesen, sodass die Eingabe gültig für den eingesetzten Datentyp ist.
- Kann nicht gültig eingelesen werden, dann wird der Variable der Nullwert zugewiesen und der Eingabestrom in einen Fehlermodus versetzt.

2.7 Rechnen mit Zahlen

Nachdem wir jetzt wissen wie die Eingabe und Ausgabe prinzipiell funktioniert, und die fundamentalen Datentypen `int` und `char` kennengelernt haben und auch die Anweisung `if` kennen, wollen wir einen kleinen Rechner entwickeln, der einen arithmetischen Ausdruck bestehend aus zwei Operanden und einem der Operatoren `+` (Addition), `-` (Subtraktion), `*` (Multiplikation), `/` Division ausrechnen kann.

Gültige Ausdrücke wären: `1 + 2` (wie gehabt), `2 - 2`, `2 * 3` oder `4 / 2`.

Wir legen fest, dass wir die Eingabe wie gehabt gestalten, also zuerst die erste Zahl eingeben, dann die Operation und danach die zweite Zahl. Das Ergebnis wird zum Schluss angezeigt.

Ein erster Versuch führt uns zu dem folgenden Quelltext in der Datei `calc.cpp`, wobei ich in diesem Fall nur mehr den Rumpf von main wiedergebe:

```
1  int num1;
2  int num2;
3  char op;  // eine Abkürzung für operator (keyword in C++!)
4
5  cout << "Die erste Zahl: ";
6  cin >> num1;
7  cout << "Operator [+,-,*,/]: ";
8  cin >> op;
9  cout << "Die zweite Zahl: ";
10 cin >> num2;
11
12 if (op == '+') {
13     cout << "Das Ergebnis ist: " << num1 + num2 << endl;
14 } else if (op == '-') {
15     cout << "Das Ergebnis ist: " << num1 - num2 << endl;
16 } else if (op == '*') {
17     cout << "Das Ergebnis ist: " << num1 * num2 << endl;
18 } else if (op == '/') {
19     cout << "Das Ergebnis ist: " << num1 / num2 << endl;
20 }
```

Bei diesem Programm verwenden wir einen neuen fundamentalen Datentyp, nämlich `char`, das für ein einzelnes Zeichen steht. Die Variable nennen wir `op`, da wir `operator` nicht verwenden können, da es sich um ein Keyword in C++ handelt. Beachte, dass Zeichenliterale des Typs `char` jeweils mit einem einfachen Hochkomma eingeschlossen sind.

Probiere das Programm gleich aus. Es funktioniert weitgehend mit gültigen Eingaben. Lediglich bei der Division kommt es zu zwei Problemen:

- Teste das Programm einmal mit der Berechnung von `1 / 0`. Es wird in etwa zu folgender Ausgabe kommen:

```
1 Die erste Zahl: 1
2 Operator [+,-,*,/]: /
3 Die zweite Zahl: 0
4 fish: Job 1, 'calc' durch Signal SIGFPE (Fliesskomma-Ausnahmefehler) beend
```

Das Programm bricht mit einer Fehlermeldung ab, weil die Division durch die Zahl 0 nicht definiert ist. Durch 0 kann man eben nicht dividieren.

Dieses Problem kannst du relativ leicht beheben, indem du `0` als Eingabe für die zweite Zahl nicht zulässt. D.h., du kannst eine Überprüfung einbauen und danach das Programm beenden. Beenden kannst du es in diesem Fall einfach mit einer `return` Anweisung. Baue die folgende `if` Anweisung direkt vor die Ausgabe des Ergebnisses der Division ein:

```
1  if (num2 == 0) {
2      cerr << "Der Divisor darf nicht 0 sein!" << endl;
3      return 1;
4  }
```

In diesen Fall wird eine Fehlermeldung auf `cerr` ausgegeben. Dabei handelt es sich um ein Objekt, das wie das Objekt `cout` funktioniert, nur dass es mit dem Kanal `stderr` verknüpft ist. Dieser Kanal wird verwendet, um Fehlermeldungen auszugeben. Die Ausgaben dieses Kanals sind ganz normal in der Shell sichtbar.

Dann wird die Funktion vorzeitig beendet und der Wert der `return` Anweisung wird als Exit-Code an das aufrufende Programm zurückgegeben.

- Teste weiters die Division mit den Zahlen `1` und `2` und du wirst das Ergebnis `0` anstatt dem Ergebnis `0.5` erhalten. Wie in den meisten Programmiersprachen üblich, wird auch in C++ zur Trennung der Nachkommastellen von den Vorkommastellen der Punkt verwendet.

  Warum also wird `0` angezeigt? Das hat damit zu tun, dass beide Operanden ganze Zahlen sind und C++ in diesem Fall auch nur mit ganzen Zahlen rechnet. D.h., dass das Ergebnis den gleichen Typ hat wie die beiden Operanden (vorausgesetzt diese haben beide denselben Typ).

  Dieses Problem lässt sich leicht beheben indem der Datentyp der beiden Zahlen zu `double` geändert wird. Bei dem fundamentalen Datentyp `double` handelt es sich um einen Typ, der Gleitkommazahlen repräsentiert:

```
1  double num1;
2  double num2;
```

Probiere das Programm wieder aus. Jetzt liefert es auch bei der Division das erwartete Ergebnis!

In diesem Zusammenhang ist es erwähnenswert, dass die Behandlung der Division durch 0 bei Gleitkommazahlen anders funktioniert als bei ganzen Zahlen. Damit du dieses andere Verhalten sehen kannst, nimm zeitweise die kürzlich hinzugefügte Überprüfung auf die Zahl 0 wieder aus dem Programm. Das geht am besten indem du die Überprüfung mit den Kommentarzeichen `/*` und `*/` umschließt:

```
1  /*
2  if (num2 == 0) {
3      cerr << "Der Divisor darf nicht 0 sein!" << endl;
4      return 1;
5  }
6  */
```

Die Ausgabe wird für die Zahlen 1 und 0 folgendermaßen aussehen:

```
1  Die erste Zahl: 1
2  Operator [+,-,*,/]: /
3  Die zweite Zahl: 0
4  Das Ergebnis ist: inf
```

Es kommt zu keinem Programmabbruch! Stattdessen ergibt die Division durch 0 einen eigenen Wert, der als „inf“ (für infinity, dt. unendlich) in der Ausgabe erscheint. Bei Division von -1 durch 0 ergibt sich „-inf“.

Wir sehen, dass es für Gleitkommazahlen einen eigenen Wert gibt, der anzeigt, dass eine Division durch 0 keinen definierten Wert hat. Mehr dazu später.

Als Nächstes wollen wir die Struktur unseres Programmes verbessern. An sich ist an der `if` Anweisung nichts auszusetzen, aber es besteht die Möglichkeit, dass wir die nächste Kontrollanweisung, nämlich die `switch` Anweisung kennenlernen. Ersetze deshalb die gesamte `if` Anweisung durch folgende Anweisungen:

```
switch (op) {
  case '+':
    res = num1 + num2;
    break;
  case '-':
    res = num1 - num2;
    break;
  case '*':
    res = num1 * num2;
    break;
  case '/':
    res = num1 / num2;
    break;
}

cout << "Das Ergebnis ist: " << res << endl;
```

Hier lernen wir eine neue zusammengesetzte Anweisung kennen, die `switch` Anweisung. Diese ist dafür gedacht, dass man eine Auswahl treffen kann. Hier wird in Abhängigkeit des Inhaltes der Variable `op` zu einem der Fälle (engl. *case*) verzweigt. In diesem Fall kommt eine weitere einfache Anweisung, die `break`-Anweisung vor. Diese bewirkt, dass die Ausführung direkt nach der `switch` Anweisung weitergeführt wird und nicht die Anweisungen des nachfolgenden `case` abgearbeitet werden.

Würden die `break` Anweisungen entfernt werden, dann würde als Ergebnis immer das Ergebnis der Division ermittelt werden. Probiere es aus!

Und wie sieht es aus, wenn wir mehrere Berechnungen hintereinander ausführen wollen? Wir programmieren eine Schleife und beginnen mit einer `while` Schleife: Der gesamte Quelltext der Eingabe, der `switch` Anweisung und auch der Ausgabe kommt in den Rumpf folgender `while` Anweisung:

```
while (true) {
    // hierher!
}
```

Übersetze dein Programm und führe es aus. Wichtig ist nur zu wissen wie du so ein Programm wieder beenden kannst. Startest du dein Programm in einer Shell, dann beendet die Tastenkombination CTRL-C in der Regel das Programm, sonst ist das Programm ganz normal über die Möglichkeiten der graphischen Oberfläche zu beenden.

Die while-Anweisung erwartet sich eine Bedingung (fundamentaler Datentyp bool) in runden Klammern und führt den Rumpf solange aus solange diese Bedingung wahr ist. In C++ gibt es zwei Wahrheitswerte, nämlich true und false. Da wir in die runden Klammern das Literal true geschrieben haben, das sich nicht ändert, wird die Schleife programmgesteuert nie abgebrochen. Das ist auch der Grund, dass das laufende Programm letztendlich mittels CTRL-C abgebrochen werden muss.

Da dies natürlich keine benutzerfreundliche Lösung ist, bauen wir das Programm folgendermaßen um. Die Zeile mit dem Schlüsselwort while wird durch die folgenden Zeilen ersetzt:

```
1  char proceed{'y'};
2
3  while (proceed == 'y') {
```

Direkt nach der Anzeige des Ergebnisses werden die folgenden Zeilen eingefügt:

```
1  cout << "Weiter? [y/n] ";
2  cin >> proceed;
3  cout << endl;
```

Der Benutzer wird gefragt, ob dieser weitermachen will und muss mit dem Zeichen y bestätigen. Gibt er ein beliebiges anderes Zeichen ein, dann bricht das Programm ab.

Wiederholen wir: Eine Zeichenkonstante ist in einfachen Hochkommas einzuschließen und kann auch nur ein Zeichen beinhalten. Eine C-String-Konstante ist in doppelte Hochkommas einzuschließen und kann beliebig viele Zeichen beinhalten.

Das funktioniert schon soweit gut, solange der Benutzer nur gültige Eingaben tätigt, wenn dieser allerdings falsche Eingaben macht, dann kann es bei dieser Art

von Programmierung ziemlich unangenehm werden. Probiere einmal aus statt einer Zahl zum Beispiel das Zeichen `a` einzugeben. Zuvor erinnere dich wie du das Programm beenden kannst.

Ich möchte noch einmal die Erklärung für dieses Verhalten liefern: Ist das Objekt `cin` im Fehlermodus, dann muss dieser Fehlermodus wieder verlassen werden, damit die nachfolgenden Operationen funktionieren. Das haben wir bis jetzt noch nicht gemacht.

Deshalb werden wir zumindest eine rudimentäre Fehlerbehandlung einführen. Ersetze die Eingabeanweisung für die Variable `num1` durch folgenden Code und verfahre analog für die Variable `num2`:

```
1  if (!(cin >> num1)) {
2      cerr << "Keine gültige Zahl. Abbruch" << endl;
3      return 1;
4  }
```

Was passiert hier?

- Der Operator `!` bildet die Negation eines Wahrheitswertes, also aus `true` wird `false` und aus `false` wird `true`.
- Der Operator `>>` liefert wieder das ursprüngliche Objekt zurück auf das er angewendet wird, also in unserem Fall `cin`. Da es sich dabei um keinen Wahrheitswert handelt, aber der Operator `!` verwendet wird, wird von C++ eine implizite Konvertierung vorgenommen (mehr dazu später). Konnte die Eingabeoperation nicht erfolgreich durchgeführt werden, dann wird der Operator `!` den Wert `true` zurückliefern. Damit wird der Programmablauf in den Rumpf der `if` Anweisung verzweigen, die Fehlerausgabe tätigen und das Programm beenden.

Natürlich ist dies auch noch keine wirklich zufriedenstellende Lösung, denn man würde den Benutzer normalerweise auffordern die Eingabe zu wiederholen, aber das verschieben wir auf später.

Da wir gerade dabei sind potenzielle Eingabefehler zu behandeln, fehlt in diesem Zusammenhang noch die Fehlerbehandlung bei falscher Operatoreingabe. Da ein einzelnes Zeichen eingelesen wird, kann der Eingabestrom nicht in einen Fehler gelangen, aber es kann sich trotzdem um ein falsches Operatorzeichen handeln. D.h., es ist weder ein `+` noch ein `-`, ein `*` oder ein `/`.

Jetzt müssen wir uns entscheiden wie in solch einem Fall vorgegangen werden soll. Konsequenterweise sollten wir das Programm ebenfalls beenden. Füge deshalb in der `switch` Anweisung vor der letzten geschwungenen Klammer folgenden Codeteil ein:

```
1  default:
2    cerr << "Kein gültiger Operator. Abbruch" << endl;
3    return 1;
```

Dieses `default:` bewirkt, dass der Programmablauf hier fortgesetzt wird, wenn keine der Alternativen zutreffen. Das setzt voraus, dass überall `break` Anweisungen eingebaut worden sind, da sonst der Programmablauf mit dem direkt nachfolgenden `case` oder letztendlich dem `default` fortgeführt wird.

Können wir dieses Programm noch verbessern? Ja, wir können. Schauen wir uns doch einmal die Schleife etwas genauer an! Zuerst setzen wir die Variable `proceed` und darauffolgend fragen wir diese im Schleifenkopf der `while` Schleife gleich wieder ab. Das müssen wir so tun, damit wir die Schleife zumindest einmal betreten. Genau für derartige Anwendungsfälle gibt es die `do-while` Anweisung!

Baue das Programm so um, dass aus der ursprünglichen Struktur

```
1  char proceed{'y'};
2
3  while (proceed == 'y') {
4      // Inhalt der Schleife
5  }
```

die folgende Struktur wird:

```
1  char proceed;
2
3  do {
4      // Inhalt der Schleife
5  } while (proceed == 'y');
```

Die Bedeutung der `do`-Anweisung ist ähnlich der `while`-Anweisung, nur dass der Schleifenrumpf mindestens einmal ausgeführt wird und dies so lange, solange die Bedingung am Ende erfüllt ist.

Zwei Dinge solltest du beachten:

- Ich habe in der neuen Struktur die Initialisierung der Variable `proceed` weggelassen, da dies jetzt nicht mehr notwendig ist. Erinnere dich aber, dass das unserer Regel widerspricht und daher nicht zu empfehlen ist. Hier habe ich dies nur zu Demonstrationszwecken eingebaut.
- Am Ende der `do` Anweisung ist nach der Bedingung ein Strichpunkt zwingend notwendig!

Die normale `while` Anweisung wird oft auch als kopfgesteuerte Schleife und die `do` Anweisung analog dazu als fußgesteuerte Schleife bezeichnet.

2.8 Zählschleife, Container, Funktion

Als Nächstes wollen wir eine kleine Applikation schreiben, die die übergebenen Kommandozeilenargumente sortiert ausgibt. Das Programm soll also folgende Funktionsweise haben:

```
$ sort Wien Amsterdam Paris Berlin Lissabon
Amsterdam
Berlin
Lissabon
Paris
Wien
```

Bei Wien, Amsterdam, Paris,... handelt es sich um die zu sortierenden Kommandozeilenparameter, die danach sortiert und zeilenweise ausgegeben werden sollen.

Beginnen wir mit dem Auslesen der Kommandozeilenparameter und der Ausgabe auf `stdout` und schreiben folgenden Code in die Datei `sort.cpp`:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(int argc, char* argv[]) {
6      for (int i{0}; i < argc; ++i) {
7          cout << argv[i] << endl;
8      }
9  }
```

Teste es und du wirst folgendes Ergebnis erhalten, wenn dein ausführbares Programm ebenfalls `sort` heißt:

```
1  $ sort a b c
2  sort
3  a
4  b
5  c
```

Du siehst, dass hier der eigentliche Programmname und alle übergebenen Kommandozeilenargumente zeilenweise ausgegeben werden.

Gut, aber was ist in diesem Programm enthalten?

- Zuerst bemerkst du, dass der Funktionskopf von `main` jetzt anders ist, da dieser jetzt zwei Parameter enthält.

  `argc` ist vom Typ `int` und enthält die Anzahl der übergebenen Kommandozeilenargumente. In unserem Fall hat `argc` mindestens den Wert 1, da der Programmname ebenfalls enthalten ist.

  `argv` ist ein Array (ein Feld), das Elemente vom Typ `char*` enthält. `char*` bedeutet Pointer (eingedeutscht von engl. *pointer*, dt. *Zeiger*) auf ein `char`. Ein Pointer ist ein Speicherelement – das die Adresse des referenzierten Speicherelementes, in unserem Fall ein `char` – enthält. Es handelt sich hierbei um die Darstellung wie C-Strings im Speicher abgelegt sind, nämlich ein Pointer, der an den Anfang eines Speicherbereiches – an die Adresse des ersten Zeichens

– zeigt, der eine Folge von Zeichen (Typ `char`) enthält, die mit einem Nullzeichen (`\0`) abgeschlossen ist. Das bedeutet, dass keine Länge eines C-Strings gespeichert wird, da das Nullzeichen das Ende kennzeichnet.

Ein Array ist ein Speicherbereich fester Größe, der hintereinander Elemente eines festgelegten Datentyps beinhaltet. Der Index in einem Array beginnt bei 0 und es wird ebenfalls keine Länge mitgespeichert. Das ist auch der Grund, warum der Parameter `argc` mit der Größe des Arrays mitgegeben wird. Die Größe eines Arrays kann sich in C++ nicht ändern.

- Im Rumpf der Funktion `main` ist die nächste Kontrollanweisung enthalten. Es handelt sich um die `for` Schleife, die in diesem Fall wie eine Zählschleife funktioniert.

  Diese `for` Schleife – in dieser Variante auch Zählschleife genannt – besteht aus dem Schlüsselwort `for`, gefolgt von einem runden Klammernpaar und danach dem eigentlichen Schleifenrumpf in geschwungenen Klammern. Innerhalb des Klammernpaares gibt es genau 3 Teile, die jeweils durch einen Strichpunkt voneinander getrennt sind.

  Der erste Teil beinhaltet die Initialisierung, die genau einmal durchgeführt, der zweite Teil beinhaltet eine Bedingung, die vor jedem Schleifendurchgang überprüft wird und der dritte Teil beinhaltet eine Anweisung, die am Ende jedes Schleifendurchganges ausgeführt wird:

  a. Bei Beginn der Ausführung der Schleife wird eine Variable `i` vom Typ `int` angelegt und initialisiert, die nur innerhalb der Schleife gültig ist und zur Verfügung steht. Anstatt `i{0}` wird oft auch `i=0` geschrieben.

  b. Danach wird die Bedingung überprüft und wenn diese zutrifft, wird der Schleifenrumpf ausgeführt.

     Der Schleifenrumpf besteht in unserem Fall aus einer bekannten Ausgabe. Neu ist allerdings wie auf die Elemente eines Arrays zugegriffen werden kann: `argv[i]` greift auf das Element mit dem Index `i` aus dem Array zu. Dabei handelt es sich bei uns um einen `char` Pointer. Der Ausgabeoperator `<<` von `cout` behandelt so einen Pointer indem alle Zeichen ab dem ersten Zeichen ausgegeben werden bis exklusive dem erkannten Nullzeichen.

  c. Am Ende des Schleifenrumpfes wird die Variable `i` um 1 erhöht (inkrementiert). Das kann in C++ mittels des Inkrementoperators `++` erreicht werden. Danach wird bei b. fortgefahren.

Das bedeutet, dass eine derartige `for` Schleife äquivalent zu folgender `while` Schleife ist:

```
1  {
2      int i{0};
3      while (i < argc) {
4          cout << argv[i] << endl;
5          ++i;
6      }
7  }
```

Wir hätten also genauso gut eine `while` Schleife programmieren können, aber eine derartige `for` Schleife ist kürzer und prägnanter.

Das ist soweit erledigt, aber wir bemerken natürlich, dass ebenfalls der Programmname ausgegeben wird. Das lässt sich leicht beheben indem die Zählvariable mit 1 anstatt 0 initialisiert wird!

Nun fehlt noch das Sortieren. Wir werden das Sortieren am Datentyp `vector` demonstrieren. Ein `vector` funktioniert in der Art wie ein Array, unterscheidet sich von einem Array jedoch dadurch, dass ein `vector` ein Objekttyp ist und außerdem in der Größe veränderlich ist.

Ein Datentyp, der prinzipiell mehrere Elemente beinhalten kann wird als ein Container bezeichnet. Der Typ `vector` stellt also einen Container dar.

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main(int argc, char *argv[]) {
7      vector<string> words;
8
9      for (int i{1}; i < argc; ++i) {
10         words.push_back(argv[i]);
11     }
12
13     for (int i{0}; i < words.size(); ++i) {
14         cout << words[i] << endl;
15     }
16 }
```

Teste es, es sollte funktionieren wie bisher. Und jetzt schauen wir uns den Sourcecode einmal genauer an:

- Zuerst muss natürlich eine neue Header-Datei eingebunden werden, da ja ein neuer Datentyp aus der Standardbibliothek verwendet wird.

- Innerhalb von `main` wird am Anfang eine Variable `words` angelegt, bei der es sich um einen Vektor handelt, der lauter `string` Objekte beinhalten kann.

  Die spitzen Klammern geben den Typ der beinhalteten Objekte an. Es handelt sich hierbei um die Verwendung eines Template - Datentyps `vector` für den man bei der Instanziierung des Templates einen spezifischen Datentyp – hier `string` – mitgeben muss. Wir werden das später noch im Detail besprechen.

- Die erste `for` - Schleife durchläuft alle Kommandozeilenargumente und hängt diese jeweils hinten mittels der Methode `push_back` an den Vektor `words` an.

  In C++ wird – wie in vielen anderen Programmiersprachen auch – eine Methode eines Objektes mittels des Punktoperators `.` aufgerufen. Daher bedeutet `word.push_back(argv[i])`, dass für den Vektor `word` die Methode

`push_back` aufgerufen wird und dieser das Argument `argv[i]` als Parameter übergeben wird.

- Die zweite Schleife gibt den Inhalt dieses Vektors wie gewohnt aus. Zwei Dinge sind hier zu beachten. Erstens die Verwendung der Methode `size`, die die aktuelle Größe des Vektors zurückgibt und die Verwendung der eckigen Klammern, um auf den Inhalt an dem angegebenen Index zuzugreifen. D.h., ein Vektor verhält sich bezüglich des Zugriffes wie ein Array.

Die Schnittstelle, die ein Objekt (oder eine andere Art von Software) anbietet, um als Programmierer mit diesem programmieren zu können, wird als API (application programming interface) bezeichnet.

Hier haben wir von dem API der Klasse `vector` kennengelernt, dass wie wir

- einen `vector` eines bestimmten Typs anlegen
- ein weiteres Element zum `vector` hinzufügen
- die Größe eines `vector`s abfragen
- auf ein einzelnes Element des `vector`s über seinen Index zugreifen

können.

`vector` bietet noch viele Methoden an, von denen wir in weiterer Folge noch etliche besprechen werden.

Jetzt fehlt nur mehr das Sortieren des Vektors. Dies ist aber einfach, da die Standardbibliothek von C++ sehr durchdacht und umfangreich ist:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  int main(int argc, char *argv[]) {
8      vector<string> words;
9
10     for (int i{1}; i < argc; ++i) {
11         words.push_back(argv[i]);
12     }
13
14     sort(words.begin(), words.end());
15
16     for (int i{0}; i < words.size(); ++i) {
17         cout << words[i] << endl;
18     }
19 }
```

Wir verwenden die vordefinierte Funktion `sort`, die mittels der Headerdatei `algorithm` eingebunden werden kann. `sort` benötigt zwei Argumente, nämlich den Beginn der zu sortierenden Werte und das Ende der zu sortierenden Werte, die jeweils über die entsprechenden Methoden vom Vektor `words` erhalten werden können (`begin()` respektive `end()`). Bei diesem „Anfang“ und dem „Ende“ handelt es sich jeweils um einen sogenannten *Iterator*, aber das werden wir uns später noch genau ansehen. Fertig!

Eine kleine Verbesserung wollen wir trotzdem noch anbringen, da wir den Aufruf der Funktion `sort` mit den beiden Parametern nicht als besonders komfortabel empfinden. Besser wäre es, wenn wir die beiden Parameter nicht mitgeben müssten. Also schreiben wir noch eine Funktion, die diesen Aufruf kapselt.

Ändere den Quellcode so ab, dass vor der Funktion `main` die folgende Funktionsdefinition kommt:

```
1  vector<string> sort(vector<string> words) {
2      sort(words.begin(), words.end());
3      return words;
4  }
```

Danach ersetze den alten Aufruf der Funktion `sort` durch den neuen Aufruf der Funktion `sort`:

```
1  words = sort(words);
```

Teste! Du solltest wieder das gleiche Ergebnis erhalten, nur hast du jetzt den Vorteil, dass du einen einfacheren Funktionsaufruf hast.

Wie funktioniert es?

- Zuerst wird eine Funktion definiert. Das funktioniert so ähnlich wie die Funktion `main`, nur dass diese Funktion einen Vektor von `string` Objekten als Parameter bekommt. In diesem konkreten Fall wird eine Kopie übergeben, die auf gewohnte Weise innerhalb der Funktion sortiert wird. Da es sich um eine Kopie handelt müssen wir diese mittels einer `return` Anweisung an den Aufrufer zurückgeben.
- In der Funktion `main` wird lediglich die Funktion aufgerufen und der Rückgabewert, der wiederum eine Kopie darstellt wird, in der Variable `words` abgelegt.

So, nun haben wir auch noch eine eigene Funktion definiert und auch aufgerufen. Dabei haben wir gelernt, dass Parameter und Rückgabewert in C++ standardmäßig als Kopie übergeben werden.

2.9 „foreach“-Schleife, Struktur, Lambdafunktion

Nehmen wir einmal an, dass wir jetzt Personen in unserem Programm speichern und sortieren wollen. Eine Person soll bei uns über einen Vornamen und einen Nachnamen sowie unter Umständen über weitere Attribute verfügen.

Es bietet sich an, diese Daten in einem benutzerdefinierten Datentyp `Person` abzulegen, der über die entsprechenden Attribute (in C++ ‚member variable‘) verfügt:

```
1  // person.cpp
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>
5
6  using namespace std;
7
8  struct Person {
9      string first_name;
10     string last_name;
11     int year_of_birth;
12 };
```

Bis jetzt handelt es sich noch um kein fertiges Programm. Lediglich der benutzerdefinierte Datentyp `Person` wurde definiert, der aus drei Instanzvariablen besteht. Ein `struct` ist in C++ genau das gleiche wie eine Klasse (`class`) nur, dass alle Attribute und Methoden defaultmäßig öffentlich zugreifbar sind (`public`).

Jetzt erweitern wir den bestehenden Code, sodass ein fertiges Programm entsteht:

```
1  int main() {
2      Person p1{"Max", "Mustermann", 1990};
3      Person p2{"Otto", "Normalverbraucher", 1950};
4      Person p3{"Susi", "Musterfrau", 1991};
5  }
```

In `main` werden jetzt drei Personen angelegt. Das Interessante daran ist, dass die einheitliche Initialisierung verwendet werden kann, um die Attribute jeweils initialisieren zu können.

Zum Speichern werden wir wieder auf unseren bewährten `vector` setzen:

```
1  vector<Person> persons{p1, p2, p3};
```

Auch hier sehen wir, dass dieser mit der einheitlichen Initialisierung sehr elegant mit Anfangswerten versehen werden kann.

Jetzt wollen wir uns an das Sortieren wagen. Aber nach welchem Kriterium soll sortiert werden und wie setzt man es um? Wir wollen nach dem Geburtsjahr sortieren und die Umsetzung sieht folgendermaßen aus:

```
1  sort(persons.begin(),
2       persons.end(),
3       [](Person p1, Person p2) {
4           return p1.year_of_birth < p2.year_of_birth;
5       });
```

Hier sehen wir, dass unsere Sortierfunktion einen dritten Parameter bekommen kann. Es handelt sich hier um eine Lambdafunktion. So eine Lambdafunktion ist eine anonyme Funktion, eine Funktion ohne Namen, die direkt bei der Verwendung definiert wird.

Ohne jetzt genau auf die Syntax eingehen zu wollen, sehen wir, dass diese mit eckigen Klammern beginnt und danach wie eine normale Funktion aussieht. Diese Funktion wird von `sort` für je zwei Elemente der zu sortierenden Datenstruktur aufgerufen, um die Reihenfolge zu bestimmen. Liefert die Lambdafunktion `true` zurück, dann soll `p1` vor `p2` angeordnet werden, ansonsten umgekehrt.

Die Ausgabe erfolgt über eine „foreach“ Schleife:

```
1  for (auto p : persons) {
2      cout << p.first_name << " " << p.last_name << " "
3           << p.year_of_birth << endl;
4  }
```

Schauen wir uns diese Schleife im Detail an:

- Der Schleifenkopf beinhaltet ein für uns neues Schlüsselwort `auto`, das in Deklarationen verwendet werden kann. Es gibt an, dass der Compiler den Typ in dieser Deklaration selber ermitteln soll. In unserem Fall soll der Typ der Laufvariable `p` selbsttätig bestimmt werden.

  Der nachfolgende Doppelpunkt bedeutet hier, dass die Schleife über alle Elemente des nachfolgenden Vektors `persons` iterieren soll: In jedem einzelnen Schleifendurchgang nimmt die Variable `p` den nächsten Wert an. Da der Typ

von `persons` der Typ `vector<Person>` ist, wird vom Compiler der Typ von `p` als `Person` bestimmt.

- Im Schleifenrumpf steht die Laufvariable `p` zur Verfügung. Mittels des Punktoperators kann auf die einzelnen Instanzvariablen von `p` zugegriffen werden.

2.10 Zusammenfassung

In diesem Kapitel haben wir die grundlegenden syntaktischen Fähigkeiten von C++ kennengelernt, um einfache textorientierte, kleine, prozedurale Programme zu erstellen.

3 Grundlagen zu den Datentypen

In diesem Kapitel erhältst du einen Überblick über die eingebauten Datentypen von C++.

Hier werden wichtige Grundlagen erklärt und mit Beispielen untermauert, sodass die weiteren Datentypen eingeordnet und verstanden werden können.

3.1 Einteilung der Datentypen

Die Datentypen von C++ können in eingebaute Datentypen (engl. *built-in*) und benutzerdefinierte Datentypen (engl. *user-defined*) eingeteilt werden.

Die eingebauten Typen unterteilen sich wiederum in die fundamentalen Datentypen und in Typen, die mittels Deklaratoroperatoren (engl. *declarator operator*) aus den fundamentalen Datentypen abgeleitet (erzeugt) werden können.

Die fundamentalen Datentypen sind:

- Boolescher Typ (engl. *boolean type*) für Wahrheitswerte: `bool`.
- Zeichentyp (engl. *character type*): `char`, `wchar_t`, `char16_t` und `char32_t`.
- Ganzzahltyp (engl. *integer type*): `short`, `int`, `long`, `long long`. Von nun an verwende ich der Einfachheit halber einfach den eingedeutschten Begriff Integer anstatt Ganzzahltyp.
- Gleitkommazahltyp (engl. *floating-point type*): `float`, `double`, `long double`.
- Der Typ `void`, der angibt, dass keine Information über den Typ vorhanden ist.

Aus diesen fundamentalen Datentypen und auch den benutzerdefinierten Datentypen können mittels der folgenden Deklaratoroperatoren neue Datentypen definiert werden:

- Zeigertyp (engl. *pointer type*), wie z.B. `int*`, der einen Zeiger (eingedeutscht *Pointer*) auf einen Integer darstellt. Ein Pointer stellt eine Adresse auf einen Speicherbereich dar. In dem konkreten Beispiel ist es eben die Adresse auf einen Speicherbereich, der als Integer vom Typ `int` betrachtet wird.

Pointer sind ein wichtiges Konzept in C++, aber die Handhabung ist nicht immer ganz einfach. Wir werden die notwendigen Grundlagen nach und nach durcharbeiten.

- Arraytyp (engl. *array type*), wie z.B. `char[10]`, das ein Array von Zeichen darstellt. Ein Array ist ein Speicherbereich fester Größe, der eine Folge von Werten eines bestimmten Typs beinhaltet. Bei `char[10]` handelt es sich um einen Speicherbereich, der genau 10 Zeichen vom Typ `char` enthält.

- Referenztyp (engl. *reference type*), wie z.B. `double&`, das eine Referenz auf einen Speicherbereich vom Typ `double` ist. Eine Referenz ist nichts anderes als ein anderer Name für eine bestimmte Speicherstelle. Mehr dazu später.

Es gibt noch ein paar wichtige Begriffe:

- Der boolesche Typ, die Zeichentypen und Integer-Typen werden als integrale Typen (engl. *integral type*) bezeichnet. Ein integraler Typ ist dadurch gekennzeichnet, dass man mit diesem rechnen kann und bitweise logische Operationen (z.B. das bitweise Oder) durchführen kann!

- Ein arithmetischer Typ ist entweder ein integraler Typ oder ein Gleitkommazahltyp. Logischerweise kann man mit einem arithmetischen Typ arithmetische Operationen durchführen. Im Gegensatz zu integralen Typen kann man Gleitkommazahlen allerdings nicht mit bitweisen logischen Operationen verwenden.

  Die unterschiedlichen arithmetischen Datentypen können in Zuweisungen und in Ausdrücken (wie z.B. Berechnungen) beliebig miteinander verknüpft werden. Dabei werden implizite Konvertierungen vorgenommen. Mehr dazu später.

Zusätzlich zu den eingebauten Datentypen können noch gänzlich neue Typen, sogenannte benutzerdefinierte Datentypen, definiert werden. Dabei handelt es sich um Aufzählungen (engl. *enumeration*) und Klassen (engl. *class*). Siehe später.

3.2 Implementierungsspezifische Aspekte

Wie schon erwähnt, gibt es etliche Teile von C++, die von der konkreten Implementierung abhängig sind, da sie nicht spezifiziert sind. Das hat seinen Grund darin, dass jede Implementierung auf ein spezielles System zugeschnitten werden kann.

Eine der Zielsetzungen von C++ ist es, Hardware-nahe Programmierung durchführen zu können. Das ist speziell in der Programmierung eingebetteter Systeme wichtig, da zum Beispiel bei der Programmierung eines Handys nicht der gleiche Prozessor vorausgesetzt werden kann wie bei der Programmierung einer Desktop-Anwendung.

So verhält es sich auch mit den Datentypen, deren exakte Größe nicht spezifiziert ist, sondern nur die Verhältnisse der Größen zueinander. So kann man sich nicht darauf verlassen, dass eine ganze Zahl aus einer bestimmten Anzahl von Bytes zusammengesetzt ist. Damit ist die größte und auch die kleinste darstellbare Zahl des Typs `int` nicht definiert! Es ist lediglich festgeschrieben, dass die Größe eines `int` mindestens so groß sein muss wie die Größe eines `char`.

Schreibe folgenden Quelltext in ein Programm `size.cpp`:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      cout << sizeof(char) << endl;
7      cout << sizeof(short) << endl;
8      cout << sizeof(int) << endl;
9      cout << sizeof(long long) << endl;
10 }
```

Bei mir unter Linux in einer 32 Bit Variante kommt es zu folgenden Ausgaben:

```
1  1
2  2
3  4
4  8
```

Wichtig zu wissen ist, dass in C++ die Speichergrößen in Vielfachen der Größe eines `char` ausgedrückt werden. Damit ist per Definition `sizeof(char)` immer 1! Damit ist auf meinem System ein `int` vier Mal so groß wie ein `char`. Wie viel Bits ein `char` wirklich hat ist damit nicht zu erfahren. Meist hat ein `char` 8 Bits,

also ein Oktett. Üblicherweise hat ein Byte die Größe eines Oktetts. Ich gehe in weiterer Folge davon aus, dass ein `char` die Größe eines Bytes aufweist.

In diesem Zusammenhang werden wir auf das Schlüsselwort `static_assert` kennenlernen, das vom Compiler selber verwendet und von diesem ausgewertet wird. Hänge die folgende Zeile an das Programm an:

```
1 static_assert(3 == sizeof(int),
2               "Größe eines ints hat nicht 3 Bytes");
```

Beim Versuch das Programm jetzt zu übersetzen, wird der *Compiler* eine Fehlermeldung liefern, da ein `int` mit an Sicherheit grenzender Wahrscheinlichkeit auf keinem System genau 3 Bytes lang ist oder genauer gesagt die dreifache Größe eines `char` aufweist. Dazu muss der Compiler zur Übersetzungszeit diese Anweisung ausführen. Dazu wertet der Compiler die Bedingung aus und wenn diese nicht erfüllt ist, dann wird letztendlich der Übersetzungsvorgang mit einer Fehlermeldung abgebrochen.

Ersetzt du diese Zeile durch die folgende Zeile, dann sollte auf einem PC mit Windows, Linux oder einem MacOSX der Compiler dies übersetzen:

```
1 static_assert(4 <= sizeof(int),
2               "int hat nicht mindestens 4 Bytes");
```

In den folgenden Abschnitten werden wir etwas genauer auf diese Problematik eingehen, werden aber nicht alle Regeln angeben, da dies einerseits in dieser Form nicht möglich ist und auch nicht den Zielsetzungen dieses Buches entspricht.

3.3 Bezeichner

Wie schon besprochen muss jeder Bezeichner (engl. *identifier*, auch Name genannt) in einem C++ Programm vor der Verwendung deklariert werden. Damit wird dem Compiler einerseits ein Name und andererseits ein zugeordneter Typ bekannt gemacht.

Die folgenden Regeln gelten für den Aufbau von Bezeichnern:

- Bezeichner bestehen in C++ aus Buchstaben, Ziffern und Unterstrichen (_), wobei ein Bezeichner nicht mit einer Ziffer beginnen darf. Groß- und Kleinbuchstaben werden unterschieden.

- Allerdings dürfen Bezeichner nicht mit einem eingebauten Schlüsselwort übereinstimmen. Da es mehr als 80 Schlüsselwörter in C++ gibt, nehme ich davon Abstand diese hier anzuführen. Ein Versuch ein Keyword als Bezeichner zu verwenden wird der Compiler zuverlässig melden.

- Beginne einen Bezeichner nicht mit einem oder mehreren Unterstrichen, wie z.B. `_tmp`!

  Die genaue Regel ist komplizierter und nachfolgend angegeben: Nicht lokale Bezeichner dürfen nicht mit genau einem Unterstrich beginnen (z.B. `_error`), da diese für die Implementierung beziehungsweise das Laufzeitsystem reserviert sind. Generell dürfen Bezeichner nicht mit zwei Unterstrichen (z.B. `__status`) oder genau einem Unterstrich gefolgt von einem Großbuchstaben (z.B. `_State`) beginnen, da auch diese reserviert sind.

Es gibt viele verschiedene Arten, wie man seine Bezeichner aufbauen kann. Ich verwende die folgende Art:

- Handelt es sich um einen benutzerdefinierten Typ, dann beginnt dieser mit einem Großbuchstaben. Setzt sich dieser Typ aus mehreren Teilwörtern zusammen, dann wird jedes Teilwort ebenfalls groß geschrieben. Ein Beispiel: `NamesDirectory`.

  Eingebaute Datentypen und Datentypen aus der Standardbibliothek von C++ beginnen mit Kleinbuchstaben. Damit ist eine klare Unterscheidung möglich.

- Handelt es sich um eine Variable oder eine Funktion, dann beginnen diese mit einem Kleinbuchstaben und die etwaigen Teilwörter sind jeweils durch einen Unterstrich (engl. *underscore*) voneinander getrennt. Ein Beispiel: `first_name`.

  Methoden (engl. *method*) werden in C++ ebenfalls als Funktionen (engl. *function*) aufgefasst. Will man sich auf Methoden beschränken, dann verwendet man den Begriff Mitgliedsfunktion, jedoch ist dies lediglich eine sperrige Übersetzung des englischen Begriffes *member functions*.

Hier noch ein weiterer Tipp im Zusammenhang mit der Wahl der Identifier: Bezeichner nur aus Großbuchstaben werden nicht empfohlen, da diese meist für Makros im Kontext des Präprozessors verwendet werden.

3.4 Deklaration und Definition

Bei einer Zuordnung von Name zu Typ handelt es sich um eine *Deklaration*. Ein Name muss im Quelltext zuerst deklariert werden bevor dieser verwendet werden kann.

Eine Deklaration wird in C++ als Anweisung betrachtet. Das hat eine Bedeutung, da damit auch der Geltungsbereich (siehe Abschnitt 3.6) und speziell die Lebenszeit eines Speicherobjektes definiert ist.

Enthält eine Deklaration alle Angaben, um den Namen zu benutzen, dann spricht man von einer *Definition*. Handelt es sich dabei um die Definition einer Variable, dann wird vom Compiler für diese Variable der Speicher reserviert. Das bedeutet, dass es sich bei einer Definition um eine spezielle Deklaration handelt. Hier einige Beispiele:

```
1  bool ready;  // Definition
2  char ch;  // Definition
3  extern int state;  // Deklaration
4  double result;  // Definition
5
6  double add(double, double);  // Deklaration
7  class User;  // Deklaration
```

Für `ready`, `ch` und `result` kennt der Compiler nicht nur den Namen und den Typ, sondern es wird vom Compiler auch Speicherplatz reserviert werden, da alle Informationen vorhanden sind, um diese Variable zu benutzen.

`extern` gibt an, dass es sich nur um eine Deklaration handelt und die Definition an anderer Stelle (meist in einer anderen Datei) erfolgen muss.

Bei `add` handelt es sich um die Deklaration einer Funktion, da nur die Informationen bezüglich der Typen der Parameter sowie des Rückgabewertes vorhanden sind, aber der Rumpf der Funktion fehlt. So eine Funktionsdeklaration wird in C++ als Prototyp bezeichnet. Damit ist es möglich, dass der Compiler Funktionsaufrufe dieser Funktion übersetzen kann, aber das Linken wird fehlschlagen, da der Linker nicht weiß welche Adresse für den Funktionsaufruf eingesetzt werden muss.

Mittels `class User` wird dem Compiler mitgeteilt, dass es sich bei `User` um einen Bezeichner handelt, der für einen benutzerdefinierten Typ steht, aber es fehlt zu diesem Zeitpunkt noch die Information wie dieser Typ konkret aussieht.

Da es sich bei Definitionen um spezielle Deklarationen handelt, werde ich in weiterer Folge Deklaration als Überbegriff verwenden.

Es kann in einem Programm mehrere Deklarationen – solange es sich nicht um Definitionen handelt – des gleichen Namens im gleichen Geltungsbereich geben, dann müssen diese allerdings identisch sein. Der folgende Quelltext ist korrekt:

```
1  extern int res;
2  extern int res;
```

Allerdings wären die folgenden Deklarationen nicht korrekt, da die Typen für den Namen `res` nicht übereinstimmen:

```
1  extern int res;
2  extern double res;
```

Im Gegensatz dazu muss es genau eine Definition eines Namens im gesamten Programm geben, außer der Bezeichner wird nicht verwendet oder es handelt sich bei der Verwendung lediglich um einen Pointer, der nicht dereferenziert wird (siehe später). Das folgende Beispiel

```
1  int res;
2  int res;
```

oder auch

```
1  int res;
2  double res;
```

sind beide syntaktisch falsch, da es eben mehrere Definitionen des Bezeichners gibt.

Deklarationen werden meist mit einem Strichpunkt abgeschlossen. Die Ausnahme bildet die Funktionsdefinition, die nach der geschlossenen geschwungenen Klammer keinen Strichpunkt hat.

3.5 Struktur einer Deklaration

Den einfachsten Aufbau einer Deklaration haben wir uns schon angesehen, nämlich Typangabe gefolgt von Bezeichner. Aber es gibt noch viel mehr. Im letzten Abschnitt haben wir schon gesehen, dass es weitere Schlüsselwörter wie `extern` gibt oder, dass es Deklaratoroperatoren gibt und vieles mehr.

Vereinfacht sieht der Aufbau einer Deklaration folgendermaßen aus:

a. Am Anfang kann ein Präfix (engl. *prefix*), wie z.B. `extern` oder `static` stehen.

b. Danach folgt verpflichtend der Basistyp wie z.B. `int`, `User` oder `vector<string>`

c. Anschließend folgt ein Deklarator. Dieser besteht entweder aus einem Namen, einem Deklaratoroperator oder beidem.

d. Optional kann für Funktionen ein Suffix (engl. *suffix*) kommen wie z.B. `noexcept`.

e. Am Ende kann optional noch eine Initialisierung oder im Falle einer Funktionsdefinition der Funktionsrumpf stehen.

Diese Struktur ist hier angeführt, sodass du einen allgemeinen Überblick bekommst. Welche Möglichkeiten es detailliert gibt und wie du diese einsetzen kannst, erfolgt in folgenden Kapiteln. Damit du dir trotzdem etwas vorstellen kannst, wie so eine Deklaration aussehen kann, hier noch ein kleines Beispiel: `static int* p{new int{3}};` Eine Erklärung dazu kommt später.

Die Verwendung von Deklaratoroperatoren ist komplex, deshalb beschränke ich mich wieder auf die notwendigen und wichtigen Elemente. Eine Erklärung der wichtigen Anwendungen dieser Deklaratoroperatoren erfolgt ebenfalls später. Als Tipp empfehle ich, die Deklaration von rechts nach links zu lesen:

- Ein Pointer wird durch `*` gekennzeichnet. Ein einfaches Beispiel ist: `int* p`. Das bedeutet, dass `p` ein Pointer auf ein `int` ist.
- Eine Referenz (konkret eine lvalue Referenz, siehe später) wird durch ein `&` gekennzeichnet. Ein Beispiel wäre: `int& current{value}`. Das bedeutet,

dass `current` ein anderer Name für die Variable `value` ist. Wird `current` verändert, wie z.B. mit `current = 3;` dann wurde eben der Speicherbereich von `value` verändert, d.h. auf den Wert 3 gesetzt.
- Ein Array wird mittels eckiger Klammern angegeben: `char first_name[20]` stellt ein Array dar, das sich aus 20 Zeichen zusammensetzt.
- Eine Funktion wird von C++ durch ein folgendes Klammernpaar erkannt, wie z.B. `int sqrt();`.

Wichtig: Außer bei Definitionen von Funktionen und Namensräumen ist eine Deklaration immer mit einem Strichpunkt abzuschließen!

Prinzipiell kann man in C++ mehrere Namen in einer Deklaration deklarieren. Das ist nicht zu empfehlen und wird hier auch nicht erklärt. Das bedeutet, dass wir jede Deklaration einer Variable in genau einer Zeile schreiben, also so:

```
1  int age;
2  int count;
```

3.6 Geltungsbereich

Nachdem wir jetzt wissen was eine Deklaration ist und außerdem den Aufbau einer Deklaration kennen, stellt sich weiters die Frage wo so eine Deklaration gültig ist. Der Geltungsbereich (engl. *scope*, eingedeutscht Scope) legt eben fest in welchem Bereich der eingeführte Bezeichner gültig ist.

Als generelle Regel lässt sich sagen, dass ein Bezeichner vom Beginn der Deklaration bis zum Ende des Blockes, in dem er deklariert wird, seine Gültigkeit behält. Von dieser Regel gibt es drei Ausnahmen:

- Bezeichner, die Instanzvariablen oder Methoden einer Klasse darstellen, sind in der gesamten Klassendeklaration gültig. Wir werden uns das später noch ansehen.

- Globale Bezeichner sind in keinem Block enthalten und behalten ihre Gültigkeit bis zum Ende der Datei. Auch das werden wir uns später noch ansehen.

- Bezeichner, die innerhalb der runden Klammern einer `for`, `while` oder `switch` Anweisung oder einer Funktionsdefinition deklariert wurden erstrecken sich vom Beginn der Deklaration des Bezeichners bis zum Ende des

folgenden Blockes. Ein Beispiel für so eine Deklaration haben wir schon im Abschnitt 2.8 auf der Seite 44 gesehen.

Bezeichner können durch andere Deklarationen überschattet werden (engl. *shadowing*). Das kann durch verschachtelte Geltungsbereiche erreicht werden. Teste dies in folgendem Programm:

```
1  // scope.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int x;
7
8  int main() {
9      cout << "global x: " << x << endl;
10
11     int x;
12
13     cout << "local x: " << x << endl;
14 }
```

Die lokale Variable `x` überschattet die globale Variable `x`. Bei mir kommt es deshalb zu folgender Ausgabe:

```
1  global x: 0
2  local x: 134514675
```

Was hier zu sehen ist:

- Zuerst wird eine globale Variable `x` definiert, aber nicht explizit initialisiert.
- In `main` erfolgt die Ausgabe der globalen Variable `x`. Es handelt sich hier noch um die globale Variable, da die Gültigkeit der lokalen Variable erst mit deren

Deklaration beginnt. Weiters sieht man, dass der Wert 0 ausgegeben wird. Das hat damit zu tun, dass globale Variablen mit dem Nullwert initialisiert werden.
- Danach folgt die Deklaration einer lokalen Variable `x`. Auch diese Variable wird nicht initialisiert. Dies sieht man gut bei der nachfolgenden Ausgabe dieser lokalen Variable: es wird der Wert der Speicherstelle ausgegeben, bei mir eben 134514675.

Hänge an das Programm die folgenden Anweisungen an:

```
1  x = 1;
2
3  {
4      int x;
5      cout << "x in block: " << x << endl;
6  }
7
8  cout << "local x (after block): " << x << endl;
```

Jetzt kommt es bei mir zu folgender Ausgabe:

```
1  global x: 0
2  local x: 134514787
3  x in block: -1076411604
4  local x (after block): 1
```

Analysieren wir diese Ausgabe:

- Wir bemerken, dass die Ausgabe einer uninitialisierten Variable nicht immer den gleichen Wert aufweisen muss!
- Als Nächstes bemerken wir, dass eine lokale Variable in einem Block eine außenliegende Variable mit dem gleichen Namen „überschattet“. Damit haben die Änderungen der lokalen Variable im Block keine Auswirkungen auf die Variable außerhalb des Blockes.

Erweitern wir das Programm nochmals durch Anhängen der folgenden Anweisungen:

```
for (int x{2}; x < 3; ++x) {
    cout << "x in statement: " << x << endl;
}

cout << "local x (after assignment): " << x << endl;
```

Teste das Programm! Du siehst, dass die Deklaration der Variable innerhalb der runden Klammern der `for` Anweisung für die gesamte `for` Anweisung gilt, aber nicht außerhalb. Sehr praktisch.

Jetzt wollen wir das Beispiel über die Geltungsbereiche abschließen, indem wir eine neue Funktion definieren und diese aufrufen. Füge deshalb die nachfolgende Funktionsdefinition vor `main` ein:

```
void func(int y) {
    cout << "x in func (global): " << x << endl;
    ++y;
    cout << "y in func (local): " << y << endl;
}
```

Hänge weiters die folgenden Anweisungen in `main` hinten an:

```
int y{1};

cout << "local y (before function call): " << y << endl;

func(y);

cout << "local y (after function call): " << y << endl;
```

Der relevante Teil der Ausgabe sieht folgendermaßen aus:

```
1  local y (before function call): 1
2  x in func (global): 0
3  y in func (local): 2
4  local y (after function call): 1
```

Hier sehen wir zwei Aspekte:

- Bei `y` innerhalb der Funktion handelt es sich um einen anderen Bezeichner als `y` innerhalb von `main`. Da eine Variable standardmäßig *kopiert* wird, hat eine Änderung innerhalb der Funktion `func` keine Auswirkung auf die Variable `y` innerhalb von `main`.
- In `func` wird mit dem Namen `x` auf die globale Variable `x` zugegriffen und nicht auf die lokale Variable `x` in der Funktion `main`.

3.7 Initialisierung

3.7.1 Arten der Initialisierung

An sich gibt es vier Möglichkeiten, wie man in C++ Variablen initialisieren kann. Hier folgen Beispiele für diese verschiedenen Varianten:

```
1  int i1{1};
2  int i2 = {1};
3  int i3 = 1;
4  int i4(1);
```

Die erste Variante wird als „uniform initialization“ also die einheitliche Initialisierung bezeichnet, da sie weitgehend überall verwendet werden kann, wie zum Beispiel bei Arrays, Strukturen, Klassen und Templates.

Die erste unterscheidet sich von der zweiten Variante geringfügig, wenn ein benutzerdefinierter Datentyp über einen Konstruktor verfügt, der mit `explicit` markiert ist. Das werden wir uns später noch ansehen.

Die erste und zweite Variante haben die wichtige Eigenschaft, dass sie keine Konvertierungen zulassen, die nicht werterhaltend sind. Werterhaltend bedeutet, dass der Wert gleich dem ursprünglichen Wert sein muss, wenn er wieder in den ursprünglichen Typ gewandelt wird.

Das folgende Beispiel demonstriert dies, da die dritte und vierte Variante nicht-werterhaltende Konvertierungen durchführen:

```
1  int i5 = 2.5;  // 2.5 hat den Typ double
2  int i6(0.5);
```

Damit wird `i5` mit dem Wert `2` und `i6` mit dem Wert `0` initialisiert. Das wird als *narrowing* (dt. einengen) bezeichnet, da diese Konvertierung nicht werterhaltend ist. Genau das ist aber hier der Fall, denn würde man den Wert von `i5` wieder in einen `double` wandeln, dann erhält man lediglich den Wert `2.0`.

Es handelt sich hierbei wahrscheinlich um einen Programmierfehler. Warum sollte man eine `int` Variable mit `2.5` initialisieren wollen? Man würde in diesem konkreten Fall doch einfach `2` schreiben, nicht wahr? Hätte man stattdessen die einheitliche Initialisierung verwendet, hätte der Compiler eine entsprechende Fehlermeldung erzeugt.

Die vierte Form hat einen weiteren Nachteil. Nehmen wir einmal an, dass wir die folgende Deklaration `int i6();` haben, mit der Absicht eine Variable `i6` vom Typ `int` anzulegen und diese mit dem Nullwert zu initialisieren. In Wirklichkeit wird der Compiler dies als eine Funktionsdeklaration interpretieren, die eine Funktion mit dem Namen `i6` einführt, die keine Parameter besitzt und einen `int` zurückliefert!

Das bedeutet, dass die dritte und die vierte Variante jeweils zu vermeiden sind, wenn sich das erreichen lässt, da es hier zu sogenannten einengenden impliziten Konvertierungen kommen kann und diese beiden Varianten von der Syntax von C++ auch nicht *überall* verwendet werden können.

3.7.2 Ausnahmen zur einheitlichen Initialisierung

Von diesen vier Möglichkeiten kann nur die erste Variante weitgehend überall eingesetzt werden. Es gibt allerdings zwei Ausnahmen:

- Wenn eine `auto` Deklaration verwendet wird. Eine `auto` Deklaration kann sehr praktisch sein, da der Compiler den Typ selbständig ermittelt. Das reduziert die Redundanz und andererseits ist es manchmal gar nicht so einfach, den richtigen Typ bei der Verwendung der Standardbibliothek herauszufinden. Schreibe das folgende Beispiel und probiere es aus:

```
1  // auto.cpp
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>
5
6  using namespace std;
7
8  int main() {
9      auto words = vector<string>{"prolog", "java", "lisp",
10                                 "python", "c++"};
11
12     sort(words.begin(), words.end());
13
14     for (int i{0}; i < words.size(); ++i) {
15         cout << words[i] << endl;
16     }
17 }
```

Hier verwenden wir eine lokale Variable `words`, die beim Anlegen mit einer Liste initialisiert wird. Diese in geschwungenen Klammern eingeschlossene Liste von Werten entspricht dem Typ `std::initializer_list` und wird in der Standardbibliothek oft verwendet und kann auch für eigene Zwecke ebenfalls verwendet werden.

Da es sich bei solch einer Initialisierungsliste um diesen Typ `std::initializer_list<int>` handelt, erhält man mit `auto i{1};` auch nicht das meist erwünschte Ergebnis, äquivalent zu `int i{1}`, sondern `std::initializer_list<int> i{1}`! Das bedeutet, dass bei Verwendung von `auto` in der Regel die Initialisierungsvariante mit `=` verwendet werden muss!

Der Rest des Beispiels ist wieder gleich. Allerdings wollen wir die Gelegenheit gleich nutzen, um die Schleife mit der Ausgabe ebenfalls umzugestalten. Ersetze dazu die Schleife durch das folgende Konstrukt und teste:

```
1  for (string elem : words) {
2      cout << elem << endl;
3  }
```

Es handelt sich hierbei um eine „for each“ Schleife, wie diese auch von anderen Programmiersprachen bekannt sein dürfte: Es wird über alle Elemente des Vektors `words` iteriert und je Schleifendurchgang erhält die Laufvariable das jeweils aktuelle Element.

So, jetzt können wir auch dieses Beispiel noch verbessern. Dem Compiler ist der Typ von `words` bekannt, nämlich `vector<string>`. Daher weiß der Compiler auch, dass der Typ von `elem` nur `string` sein kann und das wird von ihm auch überprüft. Das kannst du leicht überprüfen indem du einen anderen Typ für `elem` angibst, wie z.B. `int`. Du wirst einen Syntaxfehler erhalten.

Da dem Compiler bekannt ist, um welchen Typ es sich handelt, kann er diesen auch selbst ermitteln, womit wir wiederum das Schlüsselwort `auto` zum Einsatz bringen können:

```
1  for (auto elem : words) {
2      cout << elem << endl;
3  }
```

Nicht schlecht, oder? Verwenden wir in solch einem Fall `auto` hat das den weiteren Vorteil, dass wir den Typ von `words` z.B. auf `vector<int>` ändern könnten. Die `for` Schleife müsste nicht verändert werden!

Eine andere Variante wäre eine Referenz für die Schleifenvariable zu verwenden. Füge die folgende Schleife vor die schon bestehende Schleife ein und teste:

```
1  for (auto& elem : words) {
2      elem = elem + "!";
3  }
```

Damit wird die Schleifenvariable als neuer Name für das jeweils aktuelle Schleifenobjekt verwendet, womit das aktuelle Schleifenobjekt direkt in unserem `vector` verändert werden kann. Das bedeutet, dass das Schleifenobjekt nicht kopiert wird.

Es gibt noch einen Grund nicht kopieren zu wollen, wenn nämlich das Kopieren einen beträchtlichen Aufwand darstellt, da das zu kopierende Schleifenobjekt groß ist. Unter Umständen will man allerdings das Schleifenobjekt nicht verändern und sicherstellen, dass es zu keiner Veränderung kommt, dann

schreibt man das Schlüsselwort `const` direkt vor `auto&`. Das könnte dann folgendermaßen aussehen:

```
1  for (const auto& elem : words) {
2      cout << elem << endl;
3  }
```

- Kommen wir jetzt zum zweiten Fall wo `auto` nicht einzusetzen ist. Bei dem Typ der Variable handelt es sich um einen benutzerdefinierten Typ, der sowohl einen Konstruktor mit genau den angegebenen Initialisierungswerten hat und außerdem einen Konstruktor besitzt, der eine Initialisierungsliste als Parameter erwartet.

  Wie wir schon gesehen haben, kann man einen Vektor mithilfe einer Initialisierungsliste mit Werten initialisieren, wie z.B. `vector<int> nums{1, 2, 3, 4}`. Andererseits gibt es auch einen Konstruktor mit dem man den Vektor mit einer bestimmten Anzahl von Nullwerten initialisieren kann. Angenommen man will einen Vektor mit 10 Elementen anlegen, dann kann natürlich auch `vector<int> nums{10}` nicht funktionieren, da man damit einen Vektor mit einem Element anlegt, das den Wert 10 hat.

  In diesem Fall ist es notwendig auf die explizite Konstruktorform zurückzugreifen: `vector<int> nums(10)` legt einen Vektor mit 10 Elementen an, die jeweils mit 0 belegt sind.

  Das bedeutet, dass die Klasse `vector` sowohl einen Konstruktor hat, der sich eine Initialisierungsliste erwartet als auch einen Konstruktor, der sich eine ganze Zahl erwartet. Das Verhalten der beiden Konstruktoren ist eben unterschiedlich und genau wie im vorhergehenden Absatz beschrieben.

  Beachte, dass gerade die Notation mit den runden Klammern seine Tücken hat, die wir uns im Abschnitt **6.1** über Funktionen noch ansehen werden.

3.7.3 Fehlende Initialisierungen

Fehlt bei einer Definition einer Variable eines eingebauten Datentyps die Initialisierung, dann hängt es davon ab, um welche Art der Initialisierung es sich handelt. Bei benutzerdefinierten Datentypen wird immer der Konstruktor aufgerufen.

Es gelten die beiden nachfolgenden Regeln:

- Eine globale Variable, eine Variable aus einem Namespace, eine lokale `static` Variable oder eine `static` Member-Variable wird jeweils mit dem entsprechenden Nullwert initialisiert. Bei einem benutzerdefiniertem Datentyp entspricht dies dem Aufruf des Defaultkonstruktors (siehe Abschnitt **8.4**).

  Solch eine Initialisierung wird durchgeführt bevor die Funktion `main` gestartet wird und findet in der Reihenfolge der Deklaration statt. Zwischen verschiedenen Übersetzungseinheiten (siehe Abschnitt **7.1.1**) ist die Reihenfolge der Initialisierung nicht definiert.

- Eine lokale Variable oder ein Speicherobjekt, das am Heap angelegt wurde, wird hingegen nicht initialisiert.

3.8 Speicherobjekte und Werte

3.8.1 Speicherobjekt

Wir wollen hier die Grundlagen schaffen, um die spätere Behandlung von Referenzen besser verstehen zu können.

Dazu betrachten wir vorerst den Begriff „object“ (dt. Objekt, Speicherobjekt). Es handelt sich hierbei *nicht* um ein Objekt im Sinne der Objektorientierung! Vielmehr bezeichnet dieser Begriff in C++ einen zusammenhängenden Bereich im Speicher.

Ein Speicherobjekt liegt im Speicher, beginnt an einer bestimmten Adresse und hat eine bestimmte Größe. Die Adresse ist die eindeutige Möglichkeit auf dieses Speicherobjekt zuzugreifen.

Über das Typkonzept von C++ wird diesem Speicherobjekt eine bestimmte Bedeutung zugewiesen und damit sind wiederum die Operationen festgelegt wie mit diesem Speicherobjekt verfahren werden kann.

Haben wir zum Beispiel eine Variable vom Typ `int`, die mittels `int result;` definiert worden ist, dann kann der Compiler diese Variable in einem Speicherobjekt anlegen. Da diesem Speicherobjekt jetzt über das Typsystem der Typ `int` zugewiesen ist, sind hiermit auch die Operationen eindeutig definiert, wie auf die Speicherstelle zugegriffen werden kann. Außerdem ist natürlich auch klar, dass man daher mit den üblichen arithmetischen Operationen rechnen kann.

Jedes Objekt hat eine Lebenszeit. Das bedeutet, dass es erzeugt wird, dann wird es verwendet und am Ende wird der Speicher wieder freigegeben. Die Lebenszeit eines Objektes beginnt, wenn der Konstruktor abgeschlossen ist und endet,

wenn der Destruktor mit der Ausführung beginnt. Ein Konstruktor ist eine Funktion, die für die Initialisierung eines Objektes zuständig ist und ein Destruktor ist eine Funktion, die die Beendigungsaktionen durchführt. Hat der Typ des Objektes, wie zum Beispiel ein `int`, keinen Konstruktor oder Destruktor, dann wird dies so betrachtet als hätte dieser Typ einen leeren Konstruktor oder einen leeren Destruktor.

In C++ gibt es verschiedene Arten von Lebenszeit. Die folgenden Erläuterungen greifen bezüglich der Begriffe vor und müssen beim ersten Durcharbeiten nicht verstanden werden:

Automatic Ein Objekt mit der Lebenszeit *automatic*, wird bei der Deklaration erzeugt und beendet sich, wenn der Geltungsbereich der Deklaration endet. Lokale Variable haben diese Lebenszeit.

```
1  void f() {
2      int i{1};  // Lebenszeit beginnt
3      {
4          int j{1};  // Lebenszeit beginnt
5      }  // hier endet die Lebenszeit von j
6  }  // hier endet die Lebenszeit von i
```

Static *Static* bedeutet, dass das Objekt genau einmal initialisiert wird und seine Lebenszeit erst beim Beenden des Programmes verliert. Dazu zählen globale Variable, Variable in Namensräumen, `static` deklarierte Variable in Funktionen und `static` Member-Variablen.

```
1  // Lebenszeit beginnt mit dem Start des Programmes
2  int i;  // globale Variable; initialisiert mit 0;
3
4  int main() {
5
6  }  // hier endet die Lebenszeit von i
```

Free store Als *free store* wird in C++ der Heap bezeichnet. Objekte, die dort liegen müssen explizit angefordert und explizit wieder freigegeben werden. Dazu gibt es in C++ die Operatoren `new` und `delete`.

```
1  int* p{new int{1}};  // Lebenszeit beginnt
2
3  delete p;  // Lebenszeit endet
```

Temporary objects Temporäre Objekte (*temporary objects*) entstehen als Zwischenschritt in der Auswertung eines Ausdruckes. Ihre Lebenszeit endet entweder mit dem Ende des vollständigen Ausdruckes in dem diese erzeugt worden sind oder mit der Lebenszeit der Referenz, wenn dieses temporäre Objekt an eine Referenz gebunden worden ist.

```
1  int i;
2  int j;
3  i = (i + j) * 4;
```

Temporäre Objekte sind in diesem Beispiel der Wert von `i + j` und der Wert von `(i + j) * 4`. Der vollständige Ausdruck ist `i = (i + j) * 4`. Mit dem Ende dieses vollständigen Ausdruckes beenden sich auch die Lebenszeiten aller enthaltenen temporären Objekte. Natürlich macht es in diesem Fall keinen Sinn sich darüber Gedanken zu machen, da es unerheblich ist, wann die Lebenszeit von derartigen `int` Speicherobjekten endet. Handelt es sich aber um benutzerdefinierte Datentypen, die einen Destruktor haben, dann ist es unter Umständen sehr Wohl von Interesse zu wissen, wann der Destruktor aufgerufen wird, da dieser Destruktor Nebeneffekte haben kann.

Thread-local objects Der Vollständigkeit halber erwähne ich auch noch die Objekte, deren Lebenszeit an die Lebenszeit des Threads gebunden ist.

3.8.2 Werte

Jeder Ausdruck hat einen Wert (engl. *value*). Man unterscheidet zwei Arten von Werten: *lvalue* und *rvalue*.

Als *lvalue* wird ein Ausdruck verstanden, der auf ein Objekt verweist, das über den Ausdruck hinaus erhalten bleibt. Ein lvalue hat seinen Namen davon, dass dieser auf der linken Seite einer Zuweisung („left-hand side“, abgekürzt lhs) vorkommen kann. Es gibt allerdings auch lvalues, die nicht auf der linken Seite vorkommen können, da diese `const` sind und hiermit nicht veränderbar.

Als Faustregel gilt: Ein Ausdruck, von dem du mittels des & Operators eine Adresse ermitteln kannst, ist ein lvalue.

Das folgende Beispiel enthält auch Pointer (Zeiger), die erst später beschrieben werden:

```
1  int i{};
2  int* p{&i};  // Pointer p wird mit Adresse von i initialisiert
3  const int j{1};
4
5  i = 2;
6  *p = 3;  // Objekt auf das p verweist wird mit 3 belegt
7  j = 4; // Compilerfehler, da j konstant ist
```

Hier ist `i` ein lvalue, da `i` ein Name (auch ein Ausdruck) ist, der direkt auf ein Objekt verweist. `*p` ist ebenfalls ein lvalue, da in das Objekt der Wert 3 geschrieben wird, auf das `p` verweist. `j` ist zwar ein lvalue, kann aber nicht verändert werden, da dieser konstant ist.

Beachte, dass du die Definition der Konstante `j` auch folgendermaßen anschreiben kannst:

```
1  int const j{1};
```

Damit funktioniert die „lese von rechts nach links“-Regel in 100% der Fälle. Allerdings ist die andere Schreibweise gebräuchlicher.

Das Gegenstück zu einem lvalue ist ein *rvalue*. Ein rvalue ist ein Ausdruck, der auf ein Objekt verweist, das nicht über den Ausdruck hinaus erhalten bleibt. Vereinfacht kann man sagen, dass ein rvalue ein Ausdruck ist, der kein lvalue ist.

Betrachten wir dazu den folgenden Quelltext:

```
1  int i;
2  int j;
3
4  i = (i + j) * 4;
```

Hier handelt es sich bei dem Ergebnis von `i + j` um einen rvalue und auch das Ergebnis von `(i + j) * 4` ist ein rvalue. `i` auf der linken Seite von `=` ist jedoch ein lvalue.

Kann man von einem Ausdruck eine Adresse bestimmen, dann handelt es sich typischerweise um einen lvalue, ansonsten um einen rvalue. Das bedeutet aber auch, dass für einen gegebenen Typ `T` es sowohl lvalues von dem Typ `T` als auch rvalues von dem Typ `T` geben kann!

3.9 Konstanten

Konstanten sind Bezeichner, deren Werte nicht mehr verändert werden können. Dadurch ist es notwendig, dass Konstanten bei der Definition auch initialisiert werden müssen.

Es gibt in C++ zwei Arten wie man Konstanten definieren kann:

- Durch die Verwendung des Schlüsselwortes `const`, wie wir dies schon kennengelernt haben. Hier wollen wir den Schwerpunkt allerdings auf die Initialisierung legen.

  Die Initialisierung wird bei der Verwendung von `const` zur Laufzeit vorgenommen und kann in weiterer Folge nicht mehr verändert werden. Das folgende Beispiel demonstriert dies:

```
1  // constants.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int min;
8      int size;
9
10     cout << "Minimum: ";
11     cin >> min;
12     cout << "Größe: ";
13     cin >> size;
14
15     const int max{min + size};
16     cout << "Maximum = " << max << endl;
17     //max = 3;  // Fehler!
18 }
```

- Die zweite Möglichkeit besteht in der Verwendung des Schlüsselwortes `constexpr`, das so viel wie „constant expression“ – also konstanter Ausdruck – bedeutet.

  Der Unterschied zu `const` liegt darin, dass die Initialisierung nicht zur Laufzeit, sondern zur Übersetzungszeit vorgenommen wird. Das bedeutet, dass der Compiler den Wert der Konstante bestimmt. Dazu muss der gesamte Ausdruck der Initialisierung entweder aus Literalen, `constexpr`-Konstanten oder `constexpr`-Funktionen zusammengesetzt sein. Als Ausnahme dürfen auch `const`-Konstanten verwendet werden, soferne diese mit einem konstanten Ausdruck initialisiert worden sind.

  Hänge folgende Codezeilen an dein Programm:

```
1  const int months{12};
2  constexpr int days_per_month{30};
3  constexpr int days_per_year{months * days_per_month};
4  cout << "Ein Bankenjahr hat " << days_per_year << " Tage"
5       << endl;
```

Es gibt einen weiteren Unterschied zu der Verwendung von `const`, da es sich dabei prinzipiell nicht um Speicherobjekte handelt. Das bedeutet, dass in der obigen Ausgabe anstatt `days_per_year` vom Compiler direkt der berechnete Wert 360 eingesetzt wird.

Erst wenn wir den Adressoperator explizit verwenden, wird ein Speicherobjekt angelegt. Hänge die folgenden Codezeilen an und es wird ein Speicherobjekt angelegt. Die Ausgabe erfolgt natürlich wieder wie erwartet:

```
1  const int* p{&days_per_year};
2  cout <<  "Ein Bankenjahr hat " << *p << " Tage" << endl;
```

3.10 Implizite Konvertierungen

Dieser Abschnitt erklärt die Konvertierungen, die von C++ implizit vorgenommen werden. Da die fundamentalen Datentypen noch nicht im Detail erläutert worden sind, kann dieser Abschnitt beim ersten Lesen „überflogen“ werden. Spätestens beim Durcharbeiten der einzelnen fundamentalen Datentypen kann dieser Abschnitt zwecks genaueren Verständnis nochmals aufgesucht werden.

Von C++ werden implizite Konvertierungen selbständig vorgenommen. Nehmen wir an wir haben eine Zuweisung der Form `x = expr`, wobei es sich bei `expr` um einen arithmetischen Ausdruck handelt. Die nachfolgenden Ausführungen gelten analog auch nur für den Ausdruck `expr` alleine. C++ geht folgendermaßen vor:

a. Zuerst wird eine Aufweitung der integralen Datentypen (engl. *integral promotion*, eingedeutscht Promotion) vorgenommen. Das bedeutet, dass `int`s aus kleineren Datentypen erzeugt werden und hat den Sinn, dass die Operanden in das „natürliche“ Format für arithmetische Operationen gebracht werden.

   Im speziellen werden die folgenden Promotions durchgeführt:

- Ein `char`, `signed char`, `unsigned char`, `short int` oder `unsigned short int` wird in einen `int` konvertiert, wenn der `int` alle Werte repräsentieren kann, anderenfalls in einen `unsigned int`.
- Ein `char16_t`, `char32_t`, `wchar_t` oder ein einfacher Enumerationstyp (kein `enum class`) werden zum ersten der folgenden Typen konvertiert, der alle Werte repräsentieren kann: `int`, `unsigned int`, `long`, `unsigned long`, `unsigned long long`.
- Ein `bool` wird zu einem `int` konvertiert.

Schauen wir uns dazu wieder ein Beispiel an:

```
1  // promotions.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      char a{'0'};
8      char b{'0'};
9
10     cout << "a = " << a << "; sizeof(a) = " << sizeof(a)
11          << endl;
12     cout << "b = " << b << "; sizeof(b) = " << sizeof(b)
13          << endl;
14      cout << "a + b = " << a + b << "; sizeof(a + b) = "
15           << sizeof(a + b) << endl;
16 }
```

Als Ergebnis erhältst du dann:

```
1  a = 0; sizeof(a) = 1
2  b = 0; sizeof(b) = 1
3  a + b = 96; sizeof(a + b) = 4
```

Obwohl die Größe eines `char` per Definition in C++ immer 1 ist, ist die Größe von `a + b` gleich 4. Das liegt eben daran, dass ein `char` mittels Promotion zu

einem `int` aufgeweitet wird. Im Bereich der ganzen Zahlen wird danach die Addition durchgeführt, womit das Ergebnis dann auch die Größe eines `int`s aufweist.

Als Ergebnis kommt 96 heraus, weil das dezimale Äquivalent des Zeichens `'0'` in der ASCII Kodierung der Wert 48 ist.

b. Danach werden Konvertierungen vorgenommen, um die Typen eines Ausdruckes auf einen gemeinsamen Typ zu bringen. Die exakten Regeln sind am Besten in einer Referenz nachzulesen, aber das zugrunde liegende Prinzip ist, dass jeweils auf den nächst größeren Datentyp konvertiert wird.

Das folgende Beispiel in der Datei `conversions.cpp` soll dies demonstrieren:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      long long int ll{};
7      char c{};
8
9      cout << "sizeof(ll) = " << sizeof(ll) << endl;
10     cout << "sizeof(c) = " << sizeof(c) << endl;
11     cout << "sizeof(ll + c) = " << sizeof(ll + c) << endl;
12 }
```

Auf meinem System kommt es zu folgender Ausgabe:

```
1  sizeof(ll) = 8
2  sizeof(c) = 1
3  sizeof(ll + c) = 8
```

Du siehst, dass auf meinem System ein `long long int` die Größe 8 hat, während die Größe eines `char` eben 1 ist. Das Ergebnis der Addition hat ebenfalls die Größe 8, weil eben der kleinere Typ `char` zuerst auf einen `int` aufgeweitet

wird und danach per impliziter Konversion in einen `long long int` konvertiert worden ist.

c. Als Spezialfall von b. wird der Typ des Werts des Ausdruckes in den Typ von der Variable der linken Seite der Zuweisung gebracht. Hier muss man speziell aufpassen! Hänge folgende Codezeilen an das Programm an:

```
1  int i{};
2  i = 3.5;
3  cout << i << endl;
```

Das Programm übersetzt einwandfrei und liefert selbstverständlich den Wert 3 auf der Ausgabe. Gegen eine einengende Konvertierung bei der Initialisierung können wir uns mit der einheitlichen Initialisierung wehren, aber bei einer Zuweisung kommt es – aufgrund der Wurzeln in der Programmiersprache – zu impliziten Konvertierungen, die in der Regel nicht gewollt sind.

Leider kann es noch viel unangenehmer werden. Füge folgende Zeile an das Programm an:

```
1  char c = 128;
2  cout << c << endl;
```

Auf einem System mit 8 Bit vorzeichenbehafteten `char`s, kommt es zu einem Überlauf und zu undefiniertem Verhalten. Daher sollten derartige Konvertierungen weitgehend vermieden werden. Kann dies nicht erreicht werden, dann sollte eine explizite Konvertierung zwecks Dokumentation vorgenommen werden.

Helfen kann man sich, indem man den Compiler beauftragt, eine Warnung anzuzeigen, wenn eine derartige implizite, nicht werterhaltende Konvertierung vorgenommen wird. Für die entsprechende Option für den Compiler siehe Anhang **A.2**.

3.11 Automatische Typbestimmung

Es gibt in C++ zwei Möglichkeiten wie man den Typ im Zuge einer Deklaration von C++ bestimmen lassen kann:

- Einerseits kann das schon bekannte `auto` verwendet werden (siehe Abschnitt 3.7). Der große Vorteil tritt zutage, wenn `auto` im Zusammenhang mit Templates verwendet wird.

  In C++ gibt es – wie in vielen anderen Programmiersprachen auch – ein Iteratorkonzept. Ein Vektor kann nicht nur mittels einer Zählschleife und Zugriff über den Index oder einer „for each“ Schleife, sondern auch mittels eines Iterators durchlaufen werden.

  Teste das folgende Programm:

```
1  // iterators.cpp
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  int main() {
8      vector<int> v{1, 2, 3, 4, 5};
9
10     for (vector<int>::iterator it{v.begin()};
11          it != v.end(); ++it)
12         cout << *it << endl;
13 }
```

  Was passiert? Zuerst wird ein Vektor mit `int` Werten angelegt und initialisiert. Dann werden mittels einer `for` Schleife alle Werte des Vektors durchlaufen und ausgegeben.

  Der interessante Aspekt liegt in den Iteratoren. Am Beginn der Schleife wird ein Iterator mittels `begin()` initialisiert. Dieser Iterator zeigt an den Beginn des Vektors. Solange der Iterator nicht an das Ende zeigt, wird der Schleifenrumpf betreten und am Ende der Iterator weitergesetzt.

  Das Komplizierte an dieser Schleife ist, den korrekten Typ des Iterators zu bestimmen und anzuschreiben. Dazu muss man entweder das Wissen haben, wie dieser Typ aussieht oder die Dokumentation zurate ziehen. Unabhängig davon ist es jedoch mühselig diesen anzuschreiben. Hier kommt das Schlüsselwort `auto` gerade recht. Ändere den Schleifenkopf entsprechend ab:

```
1  for (auto it = v.begin(); it != v.end(); ++it)
```

Das sieht ja schon viel einfacher aus! Der Compiler kennt ja den Rückgabewert von `vector::begin()` und deshalb können wir den Compiler den Typ selbständig bestimmen lassen. Beachte lediglich, dass die Initialisierung mittels der geschwungenen Klammern – wie schon besprochen – durch ein Gleichheitszeichen ersetzt wurde.

- Die zweite Möglichkeit besteht darin, `decltype()` zu verwenden, das hauptsächlich bei der Deklaration von Templates Verwendung findet. Für eine Verwendung verweise ich auf Abschnitt **13.2** auf der Seite **262**.

3.12 `using` Direktive und Deklaration

Das Schlüsselwort `using` wird in drei verschiedenen Arten verwendet:

- Die erste Variante kennen wir schon (`using`-Direktive), die bewirkt, dass *alle* Bezeichner des angegebenen Namensraumes im aktuellen Geltungsbereich zugreifbar sind. Obwohl alle Bezeichner zugreifbar sind, werden diese nicht zum lokalen Scope hinzugefügt.

  Das klassische Beispiel ist `using namespace std;`, das jedoch **nie** in Headerdateien (siehe Abschnitt **7.1.2**) verwendet werden soll. Die Gefahr ist, dass der aktuelle Namensraum durch eine Unmenge von Bezeichnern überflutet wird und man nicht genau ermitteln kann, woher ein spezieller Bezeichner stammt.

  Wichtig ist, dass zwar alle Namen aus dem angegebenen Namensraum zugreifbar sind, diese aber nicht im aktuellen Bereich als deklariert gelten:

```
1  // namespace_directive.cpp
2  #include <iostream>
3
4  int main() {
5      using namespace std;
6
7      int cin{0};
8      cout << cin << endl;
9  }
```

Als Ausgabe wird erwartungsgemäß nur 0 erscheinen.

- Weiters kommt `using` in Form eines Typalias (*type alias declaration*) vor, der einen neuen Namen im bestehenden Bereich für einen bestehenden Typ deklariert. Es wird kein neuer Typ erzeugt und es kann auch kein bestehender Typ verändert werden. Es wird ein neuer Name für einen schon bestehenden Typ zum lokalen Scope hinzugefügt.

  Das macht manchmal Sinn, wenn der ursprüngliche Typname zu kompliziert oder zu lange ist. Im folgenden Beispiel ist dies zu sehen:

```
1  // using.cpp
2  #include <iostream>
3  #include <vector>
4
5  int main() {
6      using IntStack = std::vector<int>;
7
8      IntStack stack{};
9
10     stack.push_back(1);
11     stack.push_back(2);
12     stack.push_back(3);
13
14     std::cout << "top: " << stack.back() << std::endl;
15     stack.pop_back();
16     std::cout << stack.back() << std::endl;
17     stack.pop_back();
18     std::cout << "bottom: " << stack.back() << std::endl;
19     stack.pop_back();
20 }
```

Es kommt erwartungsgemäß zu folgender Ausgabe:

```
1  top: 3
2  2
3  bottom: 1
```

Die Funktionsweise ist folgende:

- Hier wird dem Typ `std::vector<int>` der neue Name `IntStack` gegeben, der in weiterer Folge zur Definition der Variable `stack` verwendet wird.
- Mittels `push_back()` kann man einem Vektor hinten Elemente anfügen.
- Der Zugriff auf das letzte Element erfolgt mittels `back()`.
- Das letzte Element kann mittels `pop_back()` aus dem Vektor entfernt werden. Diese Methode liefert keinen Wert zurück.
- Beachten muss man, dass man jetzt auf `cout` und `endl` explizit mittels dem `::` Operator zugreifen muss, da keine `using namespace std;` Direktive im Programm enthalten ist.

Beachte, dass es sich um eine Deklaration handelt und hiermit das folgende Programm der Compiler *nicht* übersetzen wird, da es zwei Deklarationen mit dem gleichen Bezeichner enthält:

```
1  // using_wrong.cpp
2  #include <iostream>
3  #include <vector>
4
5  int main() {
6      using IntStack = std::vector<int>;
7
8      using IntStack = std::vector<char>;
9  }
```

- In der Form einer `using`-Deklaration wird diese verwendet, um gezielt einen Bezeichner aus einem Namensraum im aktuellen Geltungsbereich verwenden zu können. Auch hierbei wird ein Bezeichner zum lokalen Scope hinzugefügt.

  - Einerseits handelt es sich um eine Kurzschreibweise eines Typalias. Die folgende `using`-Deklaration:

    ```
    1  using std::vector;
    ```

    ist äquivalent zu dem folgenden Typalias:

```
1  using vector = std::vector;
```

– Andererseits kann eine `using`-Deklaration auch verwendet werden, um einen Bezeichner, der keinen Typ darstellt, im aktuellen Bereich verwenden zu können:

  Im gerade besprochenen Programm, das den Typalias demonstriert hat, haben wir gesehen, dass wir jetzt explizit `std::cout` und `std::endl` verwenden mussten. Da es sich allerdings um derart häufig verwendete Objekte handelt, macht es Sinn, diese gezielt im aktuellen Geltungsbereich zu importieren.

  Füge deshalb in deiner Datei `using.cpp` *vor* dem Beginn von `main` die folgenden beiden Zeilen ein:

```
1  using std::cout;
2  using std::endl;
```

  Damit kannst du in weiterer Folge die Objekte `cout` und `endl` wieder unqualifiziert verwenden. Beachte den kleinen Unterschied, der hier absichtlich eingebaut wurde: Die beiden `using`-Deklarationen sind in der gesamten Datei verfügbar, während der Typalias `IntStack` nur in der Funktion `main` zur Verfügung steht.

4 Operatoren und fundamentale Datentypen

In diesem Kapitel wird ein Überblick über Operatoren gegeben und es werden die einzelnen fundamentalen Datentypen von C++ näher beschrieben:

- Wahrheitswerte: `bool`
- Zeichen: `char`, `wchar_t`, `char16_t`, `char32_t`
- Ganze Zahlen: `int`, `long`
- Gleitkommazahlen: `float`, `double`
- `void`

4.1 Grundlegendes zu Operatoren

Operatoren können verwendet werden, um eingebaute und benutzerdefinierte Datentypen zu verknüpfen.

Man kann Operatoren nach der Anzahl der Operanden einteilen. In C++ gibt es Operatoren, die

- einen Operanden (unär), wie z.B. `++i` oder `*p`
- zwei Operanden (binär), wie z.B. `a + b`
- drei Operanden (ternär)

benötigen.

Assoziativität

Operatoren kann man weiter einteilen in linksassoziative und rechtsassoziative Operatoren.

Bei linksassoziativen Operatoren erfolgt die Abarbeitung der Teilausdrücke von links nach rechts. Das bedeutet, dass bei dem linksassoziativen Operator `+` und

dem Ausdruck `a + b + c` die Berechnung so durchgeführt wird, als ob der Ausdruck folgende Gestalt hätte: `(a + b) + c`. Das entspricht auch dem natürlichen Verhalten des Operators +, wie dieser aus der Mathematik bekannt ist.

Natürlich ist `a + b + c` gleich dem Ergebnis von `a + (b + c)`, aber bei der Subtraktion ist dies bekannterweise nicht so:

```
1  // operators.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int i{1};
8      int j{2};
9      int k{3};
10
11     cout << "i = " << i << ", j = " << j
12          << ", k = " << k << endl;
13     cout << "i - j - k = " << i - j - k << endl;
14     cout << "(i - j) - k = " << (i - j) - k << endl;
15     cout << "i - (j - k) = " << i - (j - k) << endl;
16 }
```

Teste und betrachte das Ergebnis:

```
1  i = 1, j = 2, k = 3
2  i - j - k = -4
3  (i - j) - k = -4
4  i - (j - k) = 2
```

Bei rechtsassoziativen Operatoren wird die Berechnung von rechts nach links vorgenommen. Betrachten wir den Zuweisungsoperator (engl. *assignment operator*) und füge folgende Codezeilen an:

```
1  i = j = k;
2  cout << "i = j = k;" << endl;
3  cout << "i = " << i << ", j = " << j << ", k = "
4       << k << endl << endl;
5
6  i = 1; j = 2; k = 3;
7  i = (j = k);
8  cout << "i = (j = k);" << endl;
9  cout << "i = " << i << ", j = " << j << ", k = "
10      << k << endl << endl;
11
12 i = 1; j = 2; k = 3;
13 (i = j) = k;
14 cout << "(i = j) = k;" << endl;
15 cout << "i = " << i << ", j = " << j << ", k = "
16      << k << endl << endl;
```

Dann kommt es zu folgender Ausgabe:

```
1  i = j = k;
2  i = 3, j = 3, k = 3
3
4  i = (j = k);
5  i = 3, j = 3, k = 3
6
7  (i = j) = k;
8  i = 3, j = 2, k = 3
```

Man sieht hier deutlich den Unterschied: Beim expliziten Setzen von Klammern um `(i = j)` wird dies zuerst abgearbeitet und `i` erhält den Wert von `j`. Der Wert von `(i = j)` ist eine Referenz auf `i`, das zu diesem Zeitpunkt den Wert `2` enthält. Danach wird `i` mit dem Wert von `k` überschrieben und `j` behält den alten Wert!

Unäre Operatoren und der Zuweisungsoperator sind rechtsassoziativ, alle anderen Operatoren sind linksassoziativ.

Operatorreihenfolge

Wenn verschiedene Operatoren innerhalb eines Ausdrucks verwendet werden, dann hängt die Abarbeitung von der Priorität (Vorrang) der Operatoren ab. Es ist in der Mathematik selbstverständlich, dass die „Punktoperatoren“, wie Multiplikation und Division, vor den „Strichoperatoren“, wie Addition und Subtraktion, ausgerechnet werden. In C++ ist es genauso, aber da es eine Vielzahl von Operatoren gibt, sind diese Operatoren in einer Prioritätsfolge geordnet, die eben die Reihenfolge der Abarbeitung festlegt. Die „Punktoperatoren“ haben eine höhere Priorität als die „Strichoperatoren“.

Die folgende Liste gibt die Operatorreihenfolge *ausgewählter* Operatoren in absteigender Prioritätenreihenfolge an:

1. Klammerung `()`, Lambda-Ausdrücke `[]{}`
2. Bereichsauflösung `::`
3. Zugriff auf Mitglieder `.`, `->`
4. Index `[]`, Funktionsaufruf `()`, Postfixinkrement `++`, Postfixdekrement `--`
5. Größenoperator `sizeof`, Präfixinkrement `++`, Präfixdekrement `--`, Komplement `~`, Negation `!`, Unäres Minus `-`, Unäres Plus `+`, Adresse `&`, Dereferenzierung `*`, Objekterzeugung `new`, Objektlöschung `delete`
6. Multiplikation `*`, Division `/`, Modulo `%`
7. Addition `+`, Subtraktion `-`
8. Verschiebung links `<<` und rechts `>>`
9. Vergleich: `<`, `<=`, `>=`, `>`
10. Gleichheit `==` und Ungleichheit `!=`
11. Bitweises UND `&`
12. Bitweises exklusives ODER `^`
13. Bitweises ODER `|`
14. Logisches UND `&&`
15. Logisches ODER `||`
16. Bedingungsoperator `?:`
17. Liste `{}`, Werfen einer Exception `throw`, Zuweisung `=` `|`
18. Sequenzoperator `,`

Schauen wir uns die Funktionsweise der Prioritäten einmal mittels eines Beispiels an:

```
// precedence.cpp
#include <iostream>

using namespace std;

int main() {
    int i{1};
    int j{2};
    int k{3};

    cout << i + j * k << endl;
}
```

Die Ausgabe ist – vermutlich wie erwartet – 7, da der Multiplikationsoperator eine höhere Priorität hat als der Additionsoperator.

So weit – so gut, schauen wir uns jetzt den Inkrementoperator ++ in der Präfixvariante an. Analog dazu funktioniert der Dekrementoperator --.

Hänge die beiden folgenden Anweisungen an dein Programm an:

```
cout << i + ++j * k << endl;
cout << "j = " << j << endl;
```

Die Ausgabe wird jetzt folgendermaßen aussehen:

```
7
10
j = 3
```

Der Präfixoperator ++ hat eine höhere Priorität als die anderen Operatoren und wird deshalb zuerst ausgewertet. Außerdem bewirkt der Präfixoperator, dass der inkrementierte Wert auch sofort im Ausdruck verwendet wird. Das schlägt sich auch in der Ausgabe von 10 wieder.

Als Nächstes wollen wir die Funktion des Postfixoperators ++ kennenlernen. Füge deshalb die beiden folgenden Zeilen zu deinem Programm hinzu:

```
1  cout << i + j++ * k << endl;
2  cout << "j = " << j << endl;
```

Es wird zu folgender Ausgabe kommen:

```
1  7
2  10
3  j = 3
4  10
5  j = 4
```

Du siehst, dass das Ergebnis noch immer 10 ist, obwohl der Wert von `j` mittlerweile `4` beträgt und somit auch wirklich inkrementiert wurde. Allerdings ist der Unterschied der Postfixschreibweise zur Präfixschreibweise der, dass der alte Wert von der Variable für die Auswertung des Ausdruckes verwendet wird und nicht der inkrementierte Wert.

Ok, das Prinzip hast du verstanden, aber eine kleine Erweiterung zu unserem bestehenden Programm wollen wir noch vornehmen. Probieren wir einmal eine Kombination von der Präfixvariante und der Postfixvariante von `++`. Hänge nochmals die folgenden Zeilen an und übersetze das Programm:

```
1  cout << i + ++j++ * k << endl;
2  cout << "j = " << j << endl;
```

Es geht nicht? Der Compiler meldet, dass er sich einen lvalue (L-Wert) als Operanden erwartet! Es gibt noch einen weiteren Unterschied zwischen der Präfixvariante und der Postfixvariante: Der Präfixoperator liefert einen lvalue zurück und der Postfixoperator liefert einen rvalue!

Damit sehen wir schön, dass der Postfixoperator `++` eine höhere Priorität hat als die Präfixvariante. Er liefert einen rvalue, der jedoch natürlich nicht als Operand für den Präfixoperator dienen kann, da der Präfixoperator den Wert einer Variable inkrementieren will und ein einfacher Wert keine Variable ist.

Die Lösung besteht darin, dass Klammern gesetzt werden, sodass wir eine höhere Priorität für den Präfixoperator erzwingen. Ändere deshalb die entsprechende Codezeile folgendermaßen ab:

```
1  cout << i + (++j)++ * k << endl;
```

Die Ausgabe erscheint jetzt folgendermaßen:

```
1  7
2  10
3  j = 3
4  10
5  j = 4
6  16
7  j = 6
```

Hier siehst du, dass die runden Klammern eine höhere Priorität haben, wie wir uns das natürlich erwarten und wie es auch aus der Prioritätsreihenfolge herauszulesen ist.

In der Ausgabe erkennen wir auch sehr schön das zweimalige Inkrementieren der Variable `j`!

Es ist sinnvoll folgende Faustregel zu beachten: Verwende, wenn immer es geht, die Präfixvariante der Inkrement- oder Dekrementoperatoren, da bei diesen kein temporäres Speicherobjekt angelegt werden muss!

Auswertungsreihenfolge

Unabhängig von der Assoziativität, die die Reihenfolge der Abarbeitung bei gleichen Operatoren betrifft und der Operatorpriorität, die die Abarbeitung unterschiedlicher Operatoren regelt, ist die *Auswertung* der Teilausdrücke eines Ausdruckes. Diese Reihenfolge bei der Auswertung der Teilausdrücke ist allerdings nicht definiert!

Schauen wir uns das anhand eines Beispiels an:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int f() {
6      cout << "f()" << endl;
7      return 1;
8  }
9
10 int g() {
11     cout << "g()" << endl;
12     return 2;
13 }
14
15 int h() {
16     cout << "h()" << endl;
17     return 3;
18 }
19
20 int main() {
21     int i{};
22
23     i = f() + g() * h();
24 }
```

Bei mir sieht die Ausgabe folgendermaßen aus:

```
1  f()
2  g()
3  h()
```

Die Auswertung erfolgt aufgrund der höheren Priorität des Multiplikationsoperators, sodass zuerst das Ergebnis von `g()` mit dem Ergebnis von `h()` multipliziert wird und danach das Ergebnis von `f()` addiert wird.

Die Auswertungsreihenfolge der Teilausdrücke ist aber offensichtlich eine andere. Das muss allerdings nicht so sein, da ein anderer Compiler die Auswertung so wie

die Abarbeitung der Teilausdrücke vornehmen könnte. Die Auswertungsreihenfolge ist in C++ eben nicht definiert und man kann sich auch nicht darauf verlassen, dass diese von links nach rechts erfolgt, wie dies bei mir in diesem konkreten Fall ist.

Das Ergebnis von `i` ist in diesem Fall jedoch unabhängig von der Auswertungsreihenfolge richtig. Die Funktionen `f()`, `g()` und `h()` haben jedoch den Nebeneffekt (engl. *side effect*), dass sie jeweils eine Ausgabe tätigen und über die Reihenfolge dieser Ausgaben kann eben keine Aussage getroffen werden.

Der eigentliche Sinn einer Funktion ist, dass diese einen Rückgabewert liefert. Hat eine Funktion, jedoch eine zusätzliche Auswirkung auf ihre Umgebung, dann wird das als Nebeneffekt bezeichnet.

Man kann argumentieren, dass die Aufrufreihenfolge einer Funktion, die keine Nebeneffekte hat, kein Problem darstellt. Daher sollte man Funktionen so schreiben, dass diese keine Nebeneffekte aufweisen.

Allerdings tritt das Problem der Auswertungsreihenfolge von Teilausdrücken auch in anderen Zusammenhängen auf. Hänge folgende Zeilen wieder an deine Datei:

```
1  int v[]{9,9,9,9,9};
2  i = 1;
3  v[i] = i++;
4  cout << "v[1] = " << v[1] << endl;
5  cout << "v[2] = " << v[2] << endl;
```

Bei mir kommt es zu folgender Ausgabe:

```
1  v[1] = 9
2  v[2] = 1
```

Die Anweisung `v[i] = i++;` besitzt die beiden Teilausdrücke `v[i]` und `i++`, deren Auswertungsreihenfolge wiederum nicht definiert ist! `v[i]` liefert wiederum einen lvalue auf das Element des Arrays zurück, das verändert werden soll und `i++` inkrementiert einfach die Variable `i`. Aus diesem Grund kann es – je nach Reihenfolge der Auswertung der Teilausdrücke – zu den beiden verschiedenen Ergebnissen `v[1] == 1` und `v[2] == 1` kommen.

Die Quintessenz ist, dass man innerhalb eines Ausdruckes keine Teilausdrücke verwenden sollte, die eine (oder mehrere) gemeinsame Variable verändern.

Es gibt allerdings drei Fälle in denen die Auswertungsreihenfolge von Teilausdrücken klar definiert ist:

- Für die booleschen Operatoren `&&` und `||` ist die Auswertungsreihenfolge von links nach rechts festgelegt, denn diese funktionieren nach dem Kurzschlussprinzip (engl. *short-circuit evaluation*). Das bedeutet, dass der zweite Operand nicht mehr ausgewertet wird, wenn dies nicht mehr notwendig ist. Eine Beschreibung dieser Operatoren samt Beispielen befindet sich im Abschnitt **4.3** auf der Seite **98**.

- Der Bedingungsoperator ist der einzige Operator mit drei Operanden in C++. Er hat den folgenden Aufbau `(cond) ? expr1 : expr2`. Der Wert eines derartigen Ausdruckes ist der Wert des Teilausdruckes `expr1`, wenn die Bedingung `cond` wahr ist und anderenfalls der Wert des Teilausdruckes `expr2`.

  Zuerst wird `cond` ausgewertet und danach `expr1` oder `expr2`, nie jedoch beide.

  Schreibe folgendes kleines Testprogramm und teste:

```
1  // condop.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int age{};
8
9      cout << "Alter? ";
10     cin >> age;
11     cout << ((age >= 18) ? "Erwachsen" :
12                            "Nicht erwachsen");
13 }
```

- Der Sequenzoperator findet oft im Schleifenrumpf einer `for`-Schleife Verwendung, sonst wird dieser nicht oft verwendet. Das Prinzip ist Folgendes: Der Gesamtausdruck besteht aus mehreren Teilausdrücken, die durch je

ein Komma voneinander getrennt sind. Die einzelnen Teilausdrücke werden streng von links nach rechts abgearbeitet und der Wert des Gesamtausdruckes ist der Wert des letzten Teilausdruckes.

Hier ein Beispiel:

```
1  // commaop.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int i{1};
8      int res{};
9
10     res = i++, i /= 2, i + 2;
11     cout << res << endl;
12 }
```

Teste! Stimmt die Ausgabe meiner Beschreibung überein? Nein? Dann muss man sich wiederum die Operatorprioritäten ansehen! Der Zuweisungsoperator `=` hat eine höhere Priorität als der Sequenzoperator `,`. Das bedeutet, dass die Sequenzanweisung aus den 3 Teilausdrücken `res = i++`, `i /= 2` und `i + 2` besteht. Der Wert des gesamten Ausdruckes ist 3, der jedoch nicht verwendet wird. Natürlich macht dies keinen Sinn!

Wenn man solch eine Konstruktion wirklich verwenden will (und ich rate davon wirklich ab), dann ist der gesamte Teil rechts des `=` in runde Klammern zu setzen! In diesem Fall wirst du `3` als Ergebnis in der Ausgabe erhalten. Damit sieht man auch sehr schön, dass der Sequenzoperator streng von links nach rechts auswertet und den letzten Teilausdruck als Wert des Gesamtausdruckes nimmt.

4.2 Überladen von Operatoren

Schauen wir uns einmal das folgende uns schon bekannte Beispiel jetzt aus dem Gesichtspunkt der Operatoren genauer an:

```
1  cout << 42 << endl;
```

Unter Umständen stellst du dir jetzt die Frage wie das funktionieren kann, da der Operator `<<` ja den Verschiebeoperator für integrale Datentypen darstellt. In C++ ist es möglich, dass man weitgehend jeden Operator überladen kann.

Dieses Überladen (engl. *overloading*) bedeutet, dass ein Operator für verschiedene Typen verwendet werden kann. Zum Beispiel ist der Operator `+` von Haus aus überladen, da dieser sowohl mit dem Typ `int` als auch mit `double` umgehen kann.

Dieses Überladen kann auch auf benutzerdefinierte Datentypen vorgenommen werden, wie wir es gerade bei dem Operator `<<` im Zusammenhang mit der Ausgabe von Daten auf die Standardausgabe gesehen haben. Der Operator `<<` ist ein binärer Operator und erwartet sich daher zwei Operanden. In der überladenen Variante sind dies zum Beispiel der benutzerdefinierte Datentyp `ostream` und der fundamentale Datentyp `int`.

Man kann die meisten Operatoren in C++ überladen, jedoch nicht deren Anzahl der Parameter verändern noch deren Assoziativität oder Priorität abändern.

Wie man Operatoren selbst überlädt, werden wir uns später noch genau ansehen.

4.3 Boolescher Datentyp

Der boolesche Datentyp `bool` kann nur einen der beiden Werte `true` und `false` annehmen.

Das folgende Programm zeigt die Verwendung von `bool`:

```
1  // bool.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      bool ready{};
8      char answer{};
9
10     while (!ready) {
11         cout << "[y/n] ";
12         cin >> answer;
13         ready = (answer == 'y' || answer == 'n');
14     }
15
16     cout << "Antwort: " << answer << endl;
17 }
```

Das Programm wartet auf eine Antwort, die entweder `y` oder `n` sein muss und diese Antwort ausgibt. Es gibt lediglich ein paar interessante Aspekte:

- Die Initialisierung von `ready` ist notwendig und wird in dieser Form mit `false` vorgenommen, da `false` der „Nullwert“ von `bool` ist.
- Die Bedingung der `while`-Anweisung verwendet den Operator `!`, der der Negationsoperator ist und aus dem Wert `false` den Wert `true` macht und aus `true` den Wert `false`.

  In eine sprachliche Form gebracht bedeutet diese `while` Anweisung: Solange die Eingabe noch nicht „ready“ ist, wird der Schleifenrumpf ausgeführt.
- Nachdem die Eingabe des einzelnen Zeichens vorgenommen worden ist, wird diese überprüft, ob die Eingabe dem Wert `y` oder dem Wert `n` entspricht. Der Operator `||` ist der „ODER“ Operator, der genau dann `true` als Ergebnis liefert, wenn mindestens einer der beiden Operanden den Wert `true` hat.

  Wie schon besprochen hat der Operator `||` die Eigenschaft, dass die Operanden von links nach rechts ausgewertet werden. Hat der linke Operand schon den Wert `true` ergeben, dann wird der zweite Operand nicht mehr ausgewertet und das Ergebnis der Operation ist `true`. Im vorhergehenden Beispiel

bedeutet das, dass der rechte Teil des `||` nicht mehr überprüft wird, wenn die Variable `answer` den Wert `y` hat.

Zusätzlich zu dem `!` und dem `||` Operator gibt es auch noch den `&&` Operator, der genau dann den Wert `true` als Ergebnis liefert, wenn beide Operanden den Wert `true` haben. Auch hier werden die Operanden von links nach rechts ausgewertet. Ergibt der linke Operand den Wert `false`, dann wird der rechte Operand nicht mehr ausgewertet und das Ergebnis ist `false`.

4.3.1 Konvertierungen

Aufgrund der Möglichkeit des Mixens verschiedener arithmetischer Datentypen werden in arithmetischen und logischen Ausdrücken `bool`s immer zu `int`s konvertiert, dann die Operationen mit den konvertierten Werten durchgeführt und anschließend die Werte wieder zurück zu `bool`s konvertiert. Dabei gelten die nachfolgenden Regeln:

- `true` bekommt den Wert `1`, wenn es zu einem Integer konvertiert wird und `false` bekommt den Wert `0`.
- Jeder Integer mit dem Wert `0` wird zu `false` konvertiert, wenn ein `bool` erwartet wird, jeder Wert ungleich `0` bekommt den Wert `true`.

Schauen wir uns dazu das folgende Beispiel an:

```cpp
// bool_convert.cpp
#include <iostream>

using namespace std;

int main() {
    int i{9};
    bool a{i != 0};
    bool b{true};
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    int c{};
    c = a + b;
    cout << "a + b = " << c << endl;
    c = a - b;
    cout << "a - b = " << c << endl;
    c = a && b;
    cout << "a && b = " << c << endl;
}
```

Die Ausgabe wird folgendermaßen aussehen:

```
a = 1
b = 1
a + b = 2
a - b = 0
a && b = 1
```

Was passiert?

- Die beiden booleschen Variablen `a` und `b` werden beide mit `true` initialisiert.
- Die Ausgabe erfolgt allerdings jeweils als `1`. Das liegt daran, dass standardmäßig die Ausgabe eines booleschen Wertes mittels des `<<` Operators in Form einer ganze Zahl erfolgt. Da `1` dem Wert `true` entspricht wird eben dieser ausgegeben.

- Die fragwürdige Addition von booleschen Werten wird im Bereich der ganzen Zahlen durchgeführt. Daher kommt es zur Ausgabe von 2. Genauso sieht es mit der Subtraktion aus.
- Bei && handelt es sich wieder um den logischen short-circuit Operator „UND“, aber auch bei diesem wird jeder Operand in eine ganze Zahl gewandelt, dann das bitweise „UND“ vorgenommen. Auch aus diesem Grund kommt es zur Ausgabe von 1.

Ändere jetzt den Typ von c in einen bool um und starte das Programm nochmals. Es wird zu folgender Ausgabe kommen:

```
a = 1
b = 1
a + b = 1
a - b = 0
a && b = 1
```

Wieso? Die Konvertierungen von bool zu int und das anschließende Rechnen im Bereich der ganzen Zahlen funktioniert wie vorher, nur wird jetzt jeweils das Ganzzahlergebnis in einen booleschen Wert umgewandelt. Damit wird jeder Wert ungleich 0 zu true umgewandelt, anderenfalls zu false. Die Ausgabe erfolgt durch den << Operator, der standardmäßig den booleschen Wert jedoch als ganze Zahl ausgibt.

Die Ausgabe als true oder false lässt sich jedoch leicht erreichen, indem man einen weiteren Manipulator bei der Ausgabe verwendet. Den ersten Manipulator, den wir bis jetzt kennengelernt haben, ist std::endl, der einen Zeilenvorschub in den Ausgabestrom einfügt. Jetzt benötigen wir den std::boolalpha Manipulator, der bewirkt, dass nicht 1 respektive 0, sondern true respektive false ausgegeben wird.

Füge vor der ersten Ausgabeanweisung die folgende Anweisung ein und teste:

```
cout << boolalpha;
```

In C++ handelt es sich bei std::boolalpha um einen Manipulator, der die folgende Wirkung hat:

```
a = true
b = true
a + b = true
a - b = false
a && b = true
```

Dieser Manipulator hat seine Wirkung so lange, bis dieser wieder zurückgesetzt wird. Dies kann mittels des Manipulators `std::noboolalpha` erreicht werden. Manipulatoren müssen nicht in einer eigenen Anweisung verwendet werden, sondern können auch direkt in eine Ein- oder Ausgabeoperation eingefügt werden. Du könntest auch die Ausgabeoperationen:

```
cout << boolalpha;
cout << "a = " << a << endl;
```

durch

```
cout << "a = " << boolalpha << a << endl;
```

ersetzen. Dies wiederum liegt daran, dass der Operator `<<` auch bei einem Manipulator wieder das eigentliche Objekt zurückliefert. In unserem Fall eben `cout`.

In weiterer Folge wollen wir die Werte der Variable `a` und `b` durch den Benutzer eingeben lassen. Ersetze die Definitionen der Variablen `i`, `a` und `b` durch:

```
bool a{};
bool b{};

cout << "a = ";
cin >> a;
cout << "b = ";
cin >> b;
```

Übersetze und teste jetzt dein Programm mit verschiedenen Werten für `a` und `b`. Beachte, dass du für `a` und `b` die Zahlen `0` oder `1` eingeben kannst.

Auch wenn es sicher nicht sinnvoll ist, ist es wahrscheinlich immer noch benutzerfreundlicher, den Benutzer `true` oder `false` anstatt `1` oder `0` eingeben zu lassen. Auch das kann leicht erreicht werden indem du folgende Zeile vor der ersten Eingabeanweisung einfügst:

```
1  cin >> boolalpha;
```

Damit wird der Manipulator `boolalpha` auch für die Eingabe verwendet. Der Manipulator kann auch direkt in die Anweisung zur Eingabe des ersten Wertes – analog zur Ausgabe – eingefügt werden.

Teste das Programm gleich mit verschiedenen Werten für `a` und `b`. Du kannst auch die Operation `&&` durch `||` ersetzen, um den „ODER“-Operator zu testen.

Es gibt auch den Operator `^`, der bitweises „XOR“ von ganzen Zahlen durchführt. Da boolesche Werte in `int`s gewandelt werden, kann `^` auch für `bool`s verwendet werden. Das Ergebnis einer „XOR“ von zwei Operanden ist genau dann `true`, wenn genau ein Operand `true` ist, anderenfalls `false`. Dieser Operator hat *kein* Kurzschlussverhalten, was auch keinen Sinn ergeben würde.

Jetzt folgt noch ein Beispiel, das die implizite Konvertierung eines `bool`, einen neuen Operator für ganze Zahlen, als auch die Schlüsselwörter `break` und `continue` präsentiert, die in Schleifenanweisungen verwendet werden können:

```cpp
// countdown.cpp
#include <iostream>

using namespace std;

int main() {
    cout << "Startwert: ";
    int cnt{};
    cin >> cnt;

    if (cnt)
        cout << cnt << endl;

    while (cnt > 1) {
        --cnt;
        if (cnt > 5 && cnt % 2)
            continue;
        cout << cnt << endl;
    }
}
```

Übersetze und teste das Programm! Dieses Programm implementiert offensichtlich einen „Countdown“, wobei die genaue Funktion folgendermaßen beschrieben ist:

a. Der eingegebene Startwert wird ausgegeben.
b. Dann wird in geraden Schritten heruntergezählt, solange der aktuelle Stand des Countdown größer als Fünf ist.
c. Ab der Zahl Fünf wird jede Zahl ausgegeben bis Eins erreicht ist.

Wie ist es implementiert?

- Die erste `if` Anweisung hat eine Bedingung, die nur `cnt` enthält. An sich ist `cnt` allerdings vom Typ `int`. Aufgrund der impliziten Konvertierung wird diese Zahl allerdings als `bool` betrachtet und genau jeder Wert ungleich `0` wird

als `true` interpretiert. Das bedeutet, dass der eingegebene Startwert nur ausgegeben wird, wenn dieser ungleich `0` ist.

- Danach folgt die `while` Schleife, deren Bedingung aussagt, dass `cnt` größer als `1` sein muss.
- Die erste Anweisung im Rumpf der `while` Anweisung dekrementiert (erniedrigt um `1`) den Wert von `cnt`.
- In der Bedingung der darauffolgenden `if` Anweisung wird überprüft, ob `cnt` größer als `5` *und* `cnt` ungerade ist.

  Die erste Teilbedingung überprüft, ob `cnt` größer als `5` ist, wobei `true` oder `false` als Wert auftreten kann. Da dieser boolesche Wert weiterverwendet wird, wird dieser als `int` aufgefasst und im `&&` Ausdruck weiterverwendet.

  Der Operator `%` der zweiten Teilbedingung ist der Restoperator (auch Modulooperator genannt), der den ersten Operanden durch den zweiten Operanden dividiert und den Rest ermittelt. Dieser Rest ist wiederum eine ganze Zahl.

  Diese beiden Werte werden gemäß UND verknüpft. Wenn die Bedingung erfüllt ist, dann wird die Anweisung `continue` ausgeführt, die zur Folge hat, dass der Programmablauf bei der geschlossenen geschwungenen Klammer der direkt umschließenden Schleife – auch bei einer `for` Schleife – fortgesetzt wird.

Damit ist das Programm, so wie wir es uns vorgestellt haben, auch funktionstüchtig. Nur damit ich hier eine weitere Anweisung zeigen kann, wollen wir das Programm ein wenig verändern. Ersetze den Schleifenkopf durch folgenden Quelltext:

```
1  while (true) {
2      if (cnt <= 1)
3          break;
```

Das Programm sollte wieder genauso funktionieren. Die Bedingung des Schleifenkopfes ist immer wahr, also benötigen wir eine andere Möglichkeit die Schleife abzubrechen. Hier kommt die `break` Anweisung ins Spiel, die die direkt umschließende Schleife abbricht und direkt *nach* der geschlossenen geschwungenen Klammer im Programmablauf fortsetzt. Man sieht natürlich, dass dies hier

zu keiner Verbesserung geführt hat, da das Programm länger und nicht mehr so leicht zu lesen geworden ist.

Wir setzen gleich in diesem Sinne fort und verändern das Programm nochmals zu demonstrativen Zwecken. Ändere den Schleifenkopf nochmals ab:

```
1  while (1 != 2) {
```

Es ist klar, dass `1` immer ungleich als `2` ist und damit die Bedingung ebenfalls immer wahr ist. Abgesehen von der schlechteren Lesbarkeit hat dies allerdings keine Auswirkung auf die Laufzeit des Programmes, da der Compiler diesen Ausdruck auswertet und die Bedingung intern auf `true` abändert. Damit ist dies äquivalent zu `while (true)`.

4.3.2 Boolesche Operatoren

Aufgrund der impliziten Konvertierung zu `int` können für boolesche Werte alle arithmetischen und bitweisen Operatoren verwendet werden, die im Abschnitt **4.5.1** auf Seite **113** beschrieben sind.

An eigenen booleschen Operatoren stehen `||`, `&&` und `!` zur Verfügung.

Weiters gibt es diese Operatoren `||` und `&&` auch als Zuweisungsoperatoren `||=` und `&&=`. Zuweisungsoperatoren werden im Abschnitt **4.5.1** beschrieben.

4.4 Zeichentypen

In C++ gibt es mehrere verschiedene Datentypen für einzelne Zeichen, deren Einteilung sich prinzipiell auf die Größe bezieht:

- Der grundlegende Datentyp für Zeichen ist `char`, der in weiterer Folge noch genauer behandelt wird.
- Es gibt einen Datentyp `wchar_t`, der für Unicode-Zeichen ist. Es ist keine Größe explizit im C++ Standard für diesen Datentyp festgehalten. Dies ist abhängig von der Implementierung des Compilers und der vorhandenen Zeichensätze am jeweiligen System. Es ist lediglich definiert, dass ein `wchar_t` mindestens so groß wie ein `char` sein muss.
- Meist werden UTF-16 kodierte Unicode-Zeichen für mehrsprachige Anwendungen verwendet. C++11 unterstützt dies durch einen Datentyp, der in

der Lage ist genau ein Zeichen aus UTF-16 abzuspeichern. Der Typ heißt `char16_t`.

- Um alle definierten Unicode-Zeichen in einer Kodierung mit fester Größe abspeichern zu können, benötigt man UTF-32. C++11 bietet dafür den Datentyp `char32_t`.

Viele Anwendungen verwenden als Datentyp einfach `char`. Dieser ist allerdings im Standard nicht detailliert definiert und ist von der jeweiligen Implementierung des Compilers am jeweiligen System abhängig:

- Es ist nicht definiert, ob es sich bei diesem Datentyp um einen vorzeichenbehafteten oder einen vorzeichenlosen Datentyp handelt. Wie bitte? Wieso Vorzeichen? Es handelt sich doch um einen Zeichendatentyp! Ja, schon, aber erinnere dich, dass auch die Zeichentypen zu den integralen Datentypen gehören und deshalb in arithmetischen Operationen verwenden können. Daher muss man wissen, wie diese als Zahlen interpretiert werden.

  Aus diesem Grund gibt es auch noch die Möglichkeit, dass man explizit mit `signed char` ein Vorzeichen angeben kann oder explizit mit `unsigned char` kein Vorzeichen fordert. Das sind zwei weitere Zeichentypen, die exakt dieselbe Größe wie ein `char` haben.

  Wie kann man jetzt die eigentliche Größe eines Datentyps herausfinden? Es gibt hierzu die Header-Datei `limits`, das ein Template `numeric_limits::is_signed` zur Verfügung stellt. Erstelle den folgenden Code und teste auf deinem System:

```
1  // characters.cpp
2  #include <iostream>
3  #include <limits>
4
5  using namespace std;
6
7  int main() {
8      cout << "char hat Vorzeichen: " << boolalpha
9          << numeric_limits<char>::is_signed
10         << endl;
11 }
```

Bei mir wird `true` ausgegeben. Und bei dir?

- Es ist keine Größe definiert. Es gibt lediglich eine Mindestgröße und die ist mit 8 Bits im Standard angegeben. Mehr wird über die Größe nicht ausgesagt. Das bedeutet, dass es auch Systeme geben kann, bei denen ein `char` 16 Bits oder unter Umständen auch 32 Bits lang sein kann.

  Wie schon im Abschnitt **3.2** auf der Seite **56** beschrieben, wird diese Größe als Grundeinheit zur Bestimmung der Größe der anderen Datentypen herangezogen. Das bedeutet natürlich auch, dass es sich um die kleinste Einheit handelt!

  Du kannst die aktuelle Größe eines eingebauten Datentyps mit dem Template `numeric_limits::digits` bestimmen, das die Anzahl der Binärstellen ermittelt. Hänge dazu folgenden Code an das Programm an:

```
1  cout << "char hat Vorzeichen: " << boolalpha
2       << numeric_limits<char>::digits << endl;
```

  Teste dein Programm! Und wie sieht es aus? Du erhältst wahrscheinlich 7 und nicht 8. Das liegt daran, dass `numeric_limits::digits` die Anzahl der Bits (Binärstellen) *ohne* dem Vorzeichenbit zurückliefert! Wenn dein `char` Typ vorzeichenbehaftet ist, dann musst du noch 1 hinzuaddieren!

- Diese beiden vorherigen Einschränkungen – nämlich die Mindestgröße eines `char` von 8 Bits und die Unbestimmtheit des Vorzeichens – haben Auswirkungen über die Annahme des vorhandenen Zeichenvorrats. Damit verbleiben 128 verschiedene Zeichen, womit man sinnvollerweise nur davon ausgehen kann, dass die Ziffern, die 26 lateinischen Buchstaben und etliche Satzzeichen zur Verfügung stehen, wenn man allgemein portable Programme schreiben will.

  Daher nehmen wir für die weiteren Betrachtungen einfach an, dass wir auf einem System sind, das einen `char` Typ mit genau 8 Bit hat, der zumindest ASCII unterstützt.

  In der ASCII Kodierung sind sowohl die Buchstaben als auch die Ziffern jeweils hintereinander angeordnet. Wir nutzen dies jetzt aus, um unser Programm um die Bestimmung und Ausgabe aller Dezimalziffern, mittels einer Schleife zu erweitern:

```
1  for (int i{0}; i < 10; ++i) {
2      cout << '0' + i << endl;
3  }
```

Hier nehmen wir wiederum eine Zählschleife her, die wir von `0` bis `9` zählen lassen. Im Schleifenrumpf nutzen wir aus, dass ein `char` ein integraler Datentyp ist, der zu einem `int` konvertiert wird. Dann zählen wir unsere Schleifenvariable hinzu und geben das Ergebnis aus. Teste!

Na ja, das ist nicht das Ergebnis, das wir gerne hätten: 48, 49, 50,... Wieso das? In der ASCII Kodierung hat die Ziffer `0` eben den dezimalen Wert 48 und von dort aufsteigend folgen die restlichen Ziffern. Aufgrund der Promotion wird `char` in einen `int` gewandelt. Zu 48 wird 0,1,2... addiert und deshalb werden die Zahlen 48, 49,... ausgegeben.

Deshalb müssen wir den Typ von `int` wieder in ein `char` zurückwandeln. Dazu verwenden wir den Konvertierungsoperator `static_cast`. Ersetze dazu die Ausgabe im Schleifenrumpf durch die folgende Zeile:

```
1  cout << static_cast<char>('0' + i) << endl;
```

Dies gibt an, dass der `int`-Wert in einen `char`-Wert konvertiert werden soll. Ein `static_cast` konvertiert zwischen verwandten Typen. Kann nicht konvertiert werden, dann wird der Compiler eine Fehlermeldung liefern.

Das klingt jetzt alles sehr umständlich (was es auch ist), aber das ist der Preis, den man zu zahlen hat, wenn man eine Programmiersprache verwenden will, die von der kleinsten Prozessoreinheit bis zum Superrechner einsetzbar ist. Üblicherweise stehen die Zielplattformen fest und damit kennt man auch die Rahmenbedingungen. Damit ist die Verwendung auch schon wieder definiert und einfach.

4.4.1 Zeichenliterale

Wie schon gesehen, werden Zeichenliterale durch Zeichen, die in einfachen Hochkommas eingeschlossen sind angeschrieben. Innerhalb dieser einfachen Hochkommas hat das Zeichen `\` (engl. *backslash*) eine spezielle Bedeutung. Es wird als Escape-Zeichen verwendet, wodurch das folgende oder die folgenden Zeichen eine spezielle Bedeutung erhalten.

Das folgende Beispiel gibt fünf Wörter mit einem anschließenden Zeilenumbruch aus, der durch `'\n'` angegeben ist:

```
1  cout << "Danach beginnt eine neue Zeile!" << '\n';
```

Beachte bitte die Unterschiede zwischen dem C-String-Literal mit den doppelten Hochkommas und dem Zeichenliteral mit den einfachen Hochkommas.

Hier eine Liste der häufiger verwendeten Zeichenliterale:

`\n` Newline: Zeilenvorschub (engl. *line feed*).

`\r` Carriage return: zurück an die erste Position der Zeile

`\t` Horizontal tabulator: ein Tabulator

`\\` Ein einzelner Backslash

`\'` Ein einzelnes Hochkomma

`\ooo` Der oktale Wert für das Zeichen, wobei die oktale Zahl 1, 2 oder 3 oktale Ziffern `o` umfassen kann.

`\xhhh` Der hexadezimale Wert für das Zeichen, wobei die hexadezimale Zahl eine beliebige Anzahl an hexadezimalen Ziffern `h` umfassen kann.

Besonders `\n` wird oft verwendet, da man es auch anstatt des Manipulators `endl` verwenden kann:

```
1  cout << "Hello \n";  // als Teil des C-String-Literals
2  cout << "World!" << '\n';  // als eigenes Zeichen
```

Sowohl die Verwendung von `\n` als auch der Einsatz von `endl` bewirken in beiden Fällen einen Zeilenumbruch, wenn der Stream im Textmodus geöffnet ist. Dabei wird vom Compiler für die Plattform, für die übersetzt wird, das richtige Zeilenumbruchszeichen eingesetzt. Das ist insofern interessant als zum Beispiel unter Windows ein Zeilenumbruch durch die ASCII-Zeichenfolge `\r\n` und unter Unix nur `\n` verwendet wird.

Der Unterschied von `\n` zu `endl` ist, dass mittels `endl` auch der Puffer des Streams geleert wird (engl. *flush*), in den die Zeichen geschrieben werden, während bei

`\n` nur ein Zeilenumbruch erzeugt wird. Dieser Unterschied ist wichtig, wenn in Dateien geschrieben wird, da in diesem Fall das Leeren eines Puffers die Laufzeit durchaus erhöhen kann.

Da es in C++11 auch die Möglichkeit gibt, Zeichen aus dem Unicode Standard in verschiedenen Kodierungen darzustellen, gibt es auch eine syntaktische Möglichkeit, diese anzugeben. Schaue dir die folgenden Beispiele an:

```
1 u'\xC0DE'
2 u'\uC0DE'
3 U'\UDEADC0DE'
```

Du siehst, dass es dazu einerseits ein Präfix vor dem ersten Hochkomma gibt und andererseits innerhalb der Hochkommas analog zu `\ooo` und `\xhhh` eine weitere Notation angibt, um durch 4 oder 8 hexadezimale Ziffern ein einzelnes Zeichen aus dem Unicode Standard anzugeben. Die Form mit den 4 hexadezimalen Ziffern wie z.B. `u'\uC0DE'` ist eine Abkürzung für `U'\U0000C0DE'`.

4.5 Ganze Zahlen

Ganze Zahlen gibt es ebenfalls in verschiedenen Größen. Im Abschnitt **3.2** auf der Seite **56** wurde schon prinzipiell auf die Speichergrößen eingegangen. Es ist wichtig zu wissen, dass zwar die absoluten Speichergrößen nicht fixiert sind, sehr wohl aber die Größenverhältnisse:

`short int` Das ist im Bereich der ganzen Zahlen die prinzipiell kleinste Größe. Anstatt `short int s;` kann man einfach `short s;` schreiben. Dieser Datentyp sollte nur verwendet werden, wenn es ein API erfordert oder wenn man bezüglich der Speichergröße ganz bestimmte Anforderungen hat. Das bedeutet, dass man im Zweifelsfall die Finger davon lässt.

`int` Das ist der Standarddatentyp für ganze Zahlen! Es ist sichergestellt, dass dieser Typ mindestens die gleiche Größe hat wie ein `short`.

`long int` Das ist die nächste Größe und kann einfach als `long` verwendet werden. Der Standard schreibt vor, dass ein `long` mindestens die Größe eines `int` haben muss.

`long long int` Die größte Variante kann man abgekürzt als `long long` schreiben und muss mindestens die Größe eines `long` haben.

Ganze Zahlen sind im Gegensatz zum Zeichendatentyp immer vorzeichenbehaftet. Will man das Vorzeichen explizit angeben, dann kann man dies mit den Schlüsselwörtern `signed` beziehungsweise `unsigned` erreichen. Anstatt `signed int si;` kann auch nur `signed si;` und analog dazu kann anstatt `unsigned int ui;` auch nur `unsigned ui;` verwendet werden.

Es ist dringend zu empfehlen immer bei den vorzeichenbehafteten Typen zu bleiben, da es zu weniger Fehlern bei den Berechnungen kommen kann. Verwende einen `unsigned` Typ nur, wenn ein API es direkt verlangt oder wenn bitweise Operationen ausgeführt werden müssen!

Werden Typen mit einer definierten Größe benötigt, dann kann man Typen aus der Headerdatei `<cstdint>` verwenden wie zum Beispiel `int64_t` (genau 64 Bits), `int_least32_t` (mindestens 32 Bits), `uint16_t` (vorzeichenlos, genau 16 Bits) oder `int_fast16_t` (mindestens 16 Bits, der schnellste Ganzzahltyp mit mindestens 16 Bits).

Ein Ganzzahltyp, der nicht zum Rechnen verwendet wird, aber extrem wichtig ist, ist `size_t`, der ein implementierungsabhängiger, vorzeichenloser Typ ist, der die Größe jedes beliebigen Objektes annehmen kann. Aus diesem Grund wird dieser Typ oft verwendet, wenn die Größe eines Objekts anzugeben ist. Der Operator `new` (siehe später) erwartet sich zum Beispiel einen Wert vom Typ `size_t` und der Operator `sizeof` liefert einen Wert dieses Typs zurück.

4.5.1 Arithmetische Operatoren

Für die arithmetischen Datentypen, also für alle integralen Typen und die Gleitkommazahlen, stehen die schon bekannten Operatoren `+`, `-`, `*`, `/`, `%` und `++`, `--` zur Verfügung.

Weiters gibt es für die arithmetischen Datentypen noch die Zuweisungsoperatoren `+=`, `-=`, `*=`, `/=` und `%=`. `a += 2;` ist äquivalent zu `a = a + 2;`, wobei `a` jedoch nur ein Mal ausgewertet wird! Vollständig äquivalent zu `a++;` zu `a += 1;`.

Diese einmalige Auswertung ist im folgenden Beispiel zu sehen, wobei das Konstrukt `int&` schon im Abschnitt **3.8.2** prinzipiell besprochen worden ist. Im Zusammenhang mit dem Rückgabewert von Funktionen werden wir uns dies detailliert im Abschnitt **6.2.1** auf der Seite **173** ansehen:

```
1  // arithmeticops.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int i;
7
8  int& f() {
9      cout << "f()" << endl;
10     return i;
11 }
12
13 int main() {
14     cout << "i == " << i << endl;
15     f() = f() + 1;
16     cout << "i == " << i << endl;
17     f() += 1;
18     cout << "i == " << i << endl;
19     f()++;
20     cout << "i == " << i << endl;
21 }
```

Die Ausgabe dazu sieht folgendermaßen aus und lässt klar erkennen, wann die Funktion `f` zweimal aufgerufen wird, d.h. der Ausdruck `f()` zweimal ausgewertet wird:

```
1  i == 0
2  f()
3  f()
4  i == 1
5  f()
6  i == 2
7  f()
8  i == 3
```

Beachte, dass die Funktion `f` eine Referenz zu einer globalen Variablen zurückliefert. Das bedeutet, dass es sich bei `f()` um einen lvalue handelt!

Das Zurückliefern von Referenzen ist in den wenigsten Fällen sinnvoll und sollte nur verwendet werden, wenn man sich ganz sicher ist. Hätte man eine Funktion `g()` so definiert, dann lässt sich das Programm zwar übersetzen und ausführen, wobei das Verhalten undefiniert ist:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int& g() {
6      int i;
7      return i;
8  }
9
10 int main() {
11     cout << g() << endl;
12 }
```

Bei mir liefert das Programm folgende Ausgabe, wobei das Verhalten in C++ wie gesagt nicht definiert ist:

```
-1073792860
```

Das liegt daran, dass eine Referenz auf die lokale Variable `i` zurückgeliefert wird, die am Stack liegt und am Ende der Funktion der Speicher von `i` wieder freigegeben wird.

Die Situation verschärft sich, wenn man dieser lvalue - Referenz einen Wert zuweist:

```
1  g() = 1;
```

Damit beschreibt man einen Speicherbereich, der einem nicht mehr zur Verfügung steht. Das Verhalten ist undefiniert, aber im besten Fall wird lediglich der Wert eines anderen Speicherobjektes überschrieben.

4.5.2 Bitweise Operatoren

Für ganze Zahlen gibt es auch die bitweisen Operatoren `~`, `|`, `&` und `^`:

- `~` bezeichnet für die Negation. Die bitweise Negation wandelt jedes `0`-Bit in ein `1`-Bit und jedes `1`-Bit in ein `0`-Bit und liefert diesen Wert zurück.
- `|` steht für das inklusive Oder. Das inklusive Oder liefert ein `1`-Bit an der entsprechenden Bitstelle, wenn eines der Bits der beiden Operanden an dieser Bitstelle den Wert `1` aufweist, ansonsten wird das `0`-Bit für diese Stelle als Ergebnis produziert.
- `&` bezeichnet das Und. Beim Und wird an der entsprechenden Bitstelle des Ergebnisses nur dann ein `1`-Bit produziert, wenn beide Operanden an dieser Stelle ein `1`-Bit aufweisen, ansonsten ein `0`-Bit.
- `^` ist der Operator für das exklusive Oder. Ein exklusives Oder liefert nur dann ein `1`-Bit an der entsprechenden Bitstelle, wenn genau ein Bit der beiden Operanden an dieser Bitstelle den Wert `1` aufweist, ansonsten wird das `0`-Bit für diese Stelle als Ergebnis produziert.

Das folgende Beispiel demonstriert diese Operatoren:

```
1  // bitops.cpp
2  #include <iostream>
3  #include <bitset>
4
5  using namespace std;
6
7  int main() {
8
9      unsigned int u1{0x0F};
10     unsigned int u2{0x3C};
11
12     cout << "u1 = " << bitset<8>{u1} << endl;
13     cout << "u2 = " << bitset<8>{u2} << endl;
14     cout << "u1 | u2 = " << bitset<8>{u1 | u2} << endl;
15     cout << "u1 & u2 = " << bitset<8>{u1 & u2} << endl;
16     cout << "u1 ^ u2 = " << bitset<8>{u1 ^ u2} << endl;
17     cout << "~u1 = " << bitset<8>{~u1} << endl;
18 }
```

In der Headerdatei `<bitset>` ist der Typ `bitset` enthalten, der hier lediglich verwendet wurde, um auf einfache Art und Weise den Wert als Bitmuster *auszugeben*. Ansonsten ist dies hier nicht wichtig und es hätte genauso ein `int` verwendet werden können.

Es kommt zu folgender Ausgabe:

```
1  u1 = 00001111
2  u2 = 00111100
3  u1 | u2 = 00111111
4  u1 & u2 = 00001100
5  u1 ^ u2 = 00110011
6  ~u1 = 11110000
```

Dieses Beispiel zeigt sehr schön die bitweisen Operationen! Achte aber immer auf die Prioritäten der Operatoren oder, besser noch, setze explizit Klammern. Das folgende Beispiel zeigt dies deutlich:

```
1  if (u1 & 0x07 == 1) {
2      cout << "Bit 0 oder 1 oder 2 von u1 ist gesetzt!";
3  } else {
4      cout << "Keines der Bits 0, 1, 2 von u1 ist gesetzt!";
5  }
```

Die diesbezügliche Ausgabe wird folgendermaßen aussehen:

```
1  Keines der Bits 0, 1, 2 von u1 ist gesetzt!
```

Diese Aussage ist jedoch falsch, da das niederwertigste Bit (mit der Position 0), also das am weitesten rechts stehende Bit, in `u1` eindeutig `1` ist! Das liegt daran, dass der Gleichheitsoperator eine höhere Priorität als der Operator `&` hat. Damit wird festgestellt, dass `0x7` (also die drei niederwertigsten Bits) ungleich `1` ist und daher zu `false` evaluiert. Infolge einer impliziten Konvertierung wird `false` zu `0` gewandelt und bitweise UND verknüpft mit `u1`, womit nur `0` als Ergebnis entstehen kann. Dieses Ergebnis `0` entspricht in weiterer Folge dem booleschen Wert `false` und hiermit folgt die entsprechende Ausgabe!

Zusätzlich gibt es auch noch Verschiebeoperatoren, die in einem integralen Datentyp den enthaltenen Wert um eine bestimmte Anzahl von Bits nach links oder nach rechts verschieben und den resultierenden Wert zurückliefern. Handelt es sich um einen vorzeichenlosen Typ, dann werden rechts beziehungsweise links `0`-Bits nachgeschoben. Bei einem vorzeichenbehafteten Typ wird bei einem Rechtsverschieben das Vorzeichenbit (das höchstwertigste Bit) nachgeschoben.

Erweitere die Datei um die folgenden Zeilen und teste wiederum:

```
1  cout << "u1 << 2 = " << bitset<8>{u1 << 2} << endl;
2  cout << "u1 << 3 = " << bitset<8>{u1 << 3} << endl;
3  cout << "u1 >> 2 = " << bitset<8>{u1 >> 2} << endl;
4
5  int8_t i1{-127};
6  cout << "i1 = " << static_cast<int>(i1) << " = "
7       << bitset<8>{static_cast<uint8_t>(i1)} << endl;
8  cout << "i1 >> 2 = "
9       << bitset<8>{static_cast<uint8_t>(i1 >> 2)} << endl;
```

Die Ausgabe dieser Anweisungen sieht folgendermaßen aus:

```
u1 << 2 = 00111100
u1 << 3 = 01111000
u1 >> 2 = 00000011
i1 = -127 = 10000001
i1 >> 2 = 11100000
```

In der letzten Zeile ist auch sehr schön zu sehen, wie das Vorzeichenbit zweimalig von links nachgeschoben wurde!

Zusätzlich gibt es diese Art von Operatoren auch als Zuweisungsoperatoren: `|=`, `&=`, `^=`, `<<=` und `>>=`.

4.5.3 Literale für ganze Zahlen

An sich sind die Literale für ganze Zahlen in ihrer Grundform einfach: Die Ziffern werden hintereinander angeschrieben, optional mit einem Vorzeichen als Präfix. Der Typ des Literals wird von C++ automatisch bestimmt, wobei je nach Größe entweder ein `int`, ein `long` oder ein `long long` verwendet wird.

Allerdings gibt es abgesehen von dieser Grundform viele verschiedene Möglichkeiten, wie Zahlenliterale angeschrieben werden können:

Oktale und hexadezimale Notation Mit dem Präfix `0` kann eine oktale Zahl angeschrieben werden und mit dem Präfix `0x` eine hexadezimale Zahl. In diesem Fall funktioniert die Typbestimmung wie in der Grundform, nur wird ebenfalls versucht, den Typ in eine `unsigned` Form zu bringen.

Schreibe den folgenden Quellcode:

```
// integers.cpp
#include <iostream>

using namespace std;

int main() {
    cout << 377 << endl;
    cout << 0377 << endl;
}
```

Als Ausgabe wirst du erhalten:

```
377
255
```

In der ersten Ausgabe sieht man, dass diese gleich ist mit dem dezimalen Literal im Quellcode. Bei der zweiten Zahl handelt es sich eben um das dezimale Äquivalent der oktalen Zahl 377_8. Das bedeutet, dass eine führende Null eine Bedeutung hat!

Notation für vorzeichenlose Zahlen Mittels dem Suffix `U` (auch `u` möglich) kann explizit eine vorzeichenlose ganze Zahl angegeben werden.

Notation für `long` und `long long` Zahlen Weiters können Zahlenliterale mittels dem Suffix `L` als `long` und mit `LL` als `long long` ausgezeichnet werden. Alternativ können anstatt `L` und `LL` auch die Kleinbuchstaben `l` und `ll` verwendet werden. Allerdings rate ich davon ab, da diese im Quelltext schlechter zu erkennen sind.

Diese Angabe kann auch noch mit der Angabe für vorzeichenlose Zahlen kombinierbar, wie zum Beispiel `UL` oder `ULL`.

C++14

Es besteht auch die Möglichkeit binäre Literale anzugeben, indem man das Präfix `0b` verwendet, also z.B. `0b1001` entspricht dem dezimalen Literal `9`.

Weiters kann man Zahlenliterale mittels dem einfachen Hochkomma gliedern, sodass eine bessere Lesbarkeit gegeben ist. Die ganze Zahl für eine Million kann als `1'000'000` angeschrieben werden. Die einfachen Hochkommas haben für C++ in diesem Fall keine Bedeutung und können beliebig gesetzt werden.

C++14

Eine weitere Erweiterung ist, dass Zahlenliterale zum Zwecke der besseren Lesbarkeit mittels dem Trennzeichen `'` beliebig unterteilt werden können:

```
// digitseps.cpp
##include <iostream>

using namespace std;
```

```
int main() {
cout << 1'000'000 << endl;
cout << 0644'711 << endl;
cout << 0xFF'00 << endl;
cout << 1'234.5 << endl;
cout << 0b1111'0000 << endl;
}
```

4.6 Gleitkommazahlen

Es gibt insgesamt drei verschiedenen Typen für Gleitkommazahlen, wobei auch hier die absoluten Größen von der Implementierung abhängig sind:

`float` kennzeichnet eine Gleitkommazahl mit einfacher Genauigkeit. Diese sollte nur verwendet werden, wenn man genau weiß warum man diese einsetzen will.

`double` gibt doppelte Genauigkeit an und ist der Standard für Gleitkommazahlen. Alle Literale sind ohne weitere Kennzeichnung von diesem Typ. Ein `double` ist mindestens so groß wie ein `float`.

`long double` steht für erweiterte Genauigkeit. Es ist definiert, dass `long double` mindestens so groß ist wie `double`.

Du solltest am besten immer `double` verwenden.

Achtung, es gibt keine Promotion von `float` auf `double`, wie du im folgenden Beispiel sehen kannst!

```
1  // floatings.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      cout << sizeof(2.5F) << endl;
8      cout << sizeof(2.5) << endl;
9      cout << sizeof(2.5L) << endl;
10 }
```

Die Ausgabe auf meinem System ist:

```
4
8
12
```

Konvertierungen auf den gemeinsamen größeren Datentyp in Ausdrücken finden sehr wohl statt. Hänge folgende Anweisungen an:

```
cout << sizeof(f + d) << endl;
cout << sizeof(ld + d) << endl;
```

Die Ausgabe dieser beiden Anweisungen sieht dann folgendermaßen aus:

```
8
12
```

Damit siehst du deutlich, dass die implizite Konvertierung zum größeren Datentyp von C++ vorgenommen wird.

4.6.1 Literale für Gleitkommazahlen

Gleitkommazahlen können in der üblichen Form angeschrieben werden, wie z.B. `-2.5` oder `.1e-3` ($0.1 \cdot 10^{-3}$). Der Datentyp eines solchen Literals ist per Default `double`.

Auch hier gibt es Suffixe für die beiden anderen Gleitkommazahltypen:

Einfache Genauigkeit Mit dem Suffix `F` (oder alternativ `f`) kann ein Literal als `float` angeschrieben werden, wie zum Beispiel `1.2F`.

Erweiterte Genauigkeit Um Literale als `long double` zu kennzeichnen, wird das Suffix `L` (oder `l`) verwendet, wie zum Beispiel `1.5L`.

4.7 void

Der Datentyp `void` wird in zwei verschiedenen Arten gebraucht:

- Einerseits wird `void` als Rückgabetyp einer Funktion gebraucht, wenn diese *keinen* Wert zurückliefert. Solch eine Funktion wird in der Informatik oft auch als Prozedur (engl. *procedure*) genannt.

  Die folgende Funktion liefert keinen Rückgabewert zurück und hat auch keine formalen Parameter:

```
1  void say_hello() {
2      cout << "hello" << endl;
3  }
```

  In diesem Fall wird die Funktion beendet, wenn der Programmablauf bei der geschwungenen Klammer angekommen ist. Alternativ kann auch eine `return` Anweisung ohne Wert verwendet werden:

```
1  void say_hello() {
2      cout << "hello" << endl;
3      return;
4  }
```

- Im Zusammenhang mit Zeigern (siehe Abschnitt **5.1** auf Seite **124**) wird `void*` verwendet, um einen Zeiger auf einen Wert unbekannten Typs anzugeben. In high-level C++ ist dies zu vermeiden.

5 Pointer, Array, Referenz

In diesem Kapitel werden die einzelnen Deklaratoroperatoren beschrieben, mit denen neue Typen deklariert werden können:

- Pointer
- Array
- Referenzen

Außerdem werden wir Möglichkeiten der Standardbibliothek kennenlernen, die den Umgang mit Pointern und Arrays erleichtern.

Die Inhalte dieses Kapitels, die die Pointer und die Arrays betreffen, sind für das Verständnis extrem wichtig. Trotzdem sind die sogenannten Smart-Pointer den normalen Pointern vorzuziehen und Referenzen können für viele Anwendungen verwendet werden, für die früher Pointer eingesetzt wurden.

Arrays sollte man eigentlich in der normalen Anwendungsentwicklung wenn möglich gar nicht verwenden. Statt Arrays bietet die Standardbibliothek die Typen `vector` und `array` an.

5.1 Pointer

Wie schon erwähnt, handelt es sich bei einem Pointer um ein Speicherobjekt, das eine Speicheradresse enthält. Allerdings gibt es einen wichtigen Unterschied zu einer reinen Adresse, nämlich, dass ein Pointer einen Typ hat. Dieser Typ legt fest, wie der Speicherinhalt zu interpretieren ist, auf den dieser Pointer zeigt. Damit kann C++ auch dafür sorgen, dass keine Zugriffe mittels inkompatibler Typen erfolgen.

In C++ wird solch ein Pointer auf ein Objekt des Typs `T` als `T*` angeschrieben und, in Abgrenzung zu den Smart-Pointern, auch als roher Pointer (engl. *raw pointer*) bezeichnet.

Im Zusammenhang mit Pointern gibt es zwei wichtige Operatoren:

- Der Operator `&` ermittelt die Adresse eines Objektes. Diese Operation wird auch als Referenzieren (engl. *reference*) bezeichnet.
- Der Operator `*` ermittelt den Wert eines Zeigers. Diese Operation wird als Dereferenzieren (engl. *dereference*) bezeichnet.

Diese beiden Zeichen `&` und `*` werden in Ausdrücken verwendet. Innerhalb von Deklarationen haben diese Zeichen `&` und `*` die Bedeutung der Deklaration einer Referenz (siehe Abschnitt **5.3** auf der Seite **143**) beziehungsweise die Bedeutung eines Pointers.

Betrachte den nachfolgenden Quellcode und teste:

```
1  // pointer.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      int i{1};
8      int* p{&i};
9
10     cout << "p == " << p << ", *p == " << *p
11          << ", i == " << i << endl;
12 }
```

Die Ausgabe sieht bei mir folgendermaßen aus, wobei der Wert von `p` bei dir sehr wahrscheinlich unterschiedlich sein wird:

```
1  p == 0xbf88eea4, *p == 1, i == 1
```

Was ist passiert?

- Zuerst wird eine `int` Variable angelegt und mit `1` initialisiert. Danach wird ein Pointer `p` vom Typ `int*` (Pointer auf `int`) angelegt und mit der Adresse von `i` initialisiert.

- Danach wird die Adresse von `p`, der Wert der Speicherzelle auf den `p` zeigt und der Wert von `i` ausgegeben. Die Adresse wird offensichtlich in hexadezimaler

Form ausgegeben. Da `p` auf `i` zeigt, sind die beiden letzten Werte natürlich identisch.

Füge jetzt die folgenden Zeilen anschließend an:

```
1  *p = 2;
2  cout << "*p = 2;" << endl;
3  cout << "p == " << p << ", *p == " << *p
4       << ", i == " << i << endl;
```

Starte das Programm nochmals. Bei mir kommt es zu folgender Ausgabe:

```
1  p == 0xbfc8e728, *p == 1, i == 1
2  *p = 2;
3  p == 0xbfc8e728, *p == 2, i == 2
```

Wir können folgendes bemerken:

- Die Adressen der Variable sind je Programmlauf unterschiedlich.
- Ändere ich den Wert der Speicherzelle, auf die `p` zeigt, dann ändert sich klarerweise in unserem Fall auch der Wert von `i`, da `p` die Adresse von `i` enthält.

Klar soweit? Gut, dann hänge die folgenden Zeilen wiederum hinten an:

```
1  int j{};
2  p = &j;
3  cout << "p = &j;" << endl;
4  cout << "p == " << p << ", *p == " << *p << ", i == "
5       << i << ", j == " << j << endl;
```

Starte das Programm und schaue dir die Ausgabe an.

```
p == 0xbfb65988, *p == 1, i == 1
*p = 2;
p == 0xbfb65988, *p == 2, i == 2
p = &j;
p == 0xbfb65984, *p == 0, i == 2, j == 0
```

Interpretieren wir jetzt einmal die Ausgabe:

- Der erste Teil wurde schon besprochen. Wir können in der letzten Ausgabe erkennen, dass die gespeicherte Adresse jetzt unterschiedlich ist. Das ist klar, da die Variablen `i` und `j` sicher nicht an derselben Adresse liegen werden.
- Weiters sehen wir, dass der Wert der Speicherzelle, auf die `p` zeigt, jetzt gleich dem Wert von `j` ist.

Schließen wir das Beispiel jetzt noch ab, indem wir zeigen, dass die Typen der Pointer von C++ auf Kompatibilität geprüft werden. Hänge dazu folgende Zeilen an und übersetze das Programm:

```
char c;
p = &c;
```

Ok, das lässt sich natürlich nicht übersetzen. Da der Typ von `p` eben `int*` ist und der Typ von `&c` eben `char*` ist. Das sind inkompatible Typen und deshalb wird dies vom Compiler zurückgewiesen. Beachte, dass ein `char` implizit in ein `int` konvertiert werden würde, aber dies trifft sinnvollerweise nicht auf Pointer zu: Ein Pointer auf ein `char` wird eben nicht implizit auf einen Pointer auf ein `int` konvertiert.

Ersetze diese beiden Zeilen, die sich nicht übersetzen lassen, durch folgende Zeilen:

```
p = nullptr;
cout << "p == " << p << endl;
cout << "*p == " << *p << endl;
```

Nach erfolgreicher Übersetzung kommt es beim Programmlauf zu folgender Ausgabe:

```
1  p == 0
2  Speicherzugriffsfehler
```

- Zuerst sehen wir, dass der Wert von `p` mit `0` ausgegeben wird. Das Schlüsselwort `nullptr` bezeichnet einen Pointer, der die Adresse 0 gespeichert hat.
- Danach erfolgt bei mir die Ausgabe „Speicherzugriffsfehler“. Diese Ausgabe kann bei dir anders aussehen, aber eine Fehlermeldung kommt auf jedem Fall, da kein Objekt an der Adresse 0 liegen kann. Erfolgt ein Zugriff auf die Adresse 0 wird vom Prozessor der Zugriff erkannt und der Prozess wird abgebrochen.

Jetzt stellt sich vielleicht die Frage, ob man direkt den Wert `0` in eine Pointer-Variable speichern kann. Ja, das geht aus Kompatibilitätsgründen zu älteren Versionen von C++. Ich empfehle es aber nicht zu tun, da `nullptr` leichter zu lesen ist und der Compiler in anderen Fällen (überladenen Funktionen) keine Typfehler feststellen kann, wenn `0` verwendet wird.

Meist werden Pointer in Zusammenhang mit dynamisch angeforderten Speicherobjekten verwendet. Dynamisch angeforderte Speicherobjekte werden am Heap mittels `new` angelegt und mittels `delete` wieder freigegeben. Solche Speicherobjekte haben die Lebenszeit „free store“.

```
1  p = new int{3};
2  cout << "p = new int{3};" << endl;
3  cout << "p == " << p << ", *p == " << *p << endl;
4  delete p;
```

Teste wieder. Bei mir kommt es zu folgender Ausgabe:

```
1  p = new int{3};
2  p == 0x9f32008, *p == 3
```

Wie funktionieren diese Anweisungen?

- Mit `p = new int{3};` wird ein neues `int` von der Speicherverwaltung am Heap angefordert und die Adresse davon zurückgegeben. Diese Adresse wird dann in `p` gespeichert.
- Mittels `*p` kann wiederum auf den Inhalt zugegriffen werden.
- Mit `delete p` wird der angeforderte Speicher wieder freigegeben.

5.1.1 Pointer und Strukturen

Meist wird nicht ein kleines Objekt eines fundamentalen Datentyps am Heap abgelegt, sondern benutzerdefinierte Objekte. Wir haben mit `struct` schon eine Möglichkeit kennengelernt, einen benutzerdefinierten Typ zu definieren.

Wollen wir als bekanntes Beispiel eine Person am Heap anlegen und über den Pointer auf ein Attribut der Klasse `Person` zugreifen, dann könnte das folgendermaßen aussehen:

```
1  // pointer_struct.cpp
2  #include <iostream>
3  #include <memory>
4
5  using namespace std;
6
7  struct Person {
8      string first_name;
9      string last_name;
10     int year_of_birth;
11 };
12
13 int main() {
14     Person* p{new Person{"Max", "Mustermann", 1990}};
15
16     cout << *p.first_name << endl;
17 }
```

Es war zwar gut gemeint, allerdings lässt sich das Programm nicht übersetzen. Das liegt an den Operatorprioritäten! Der Punktoperator hat höhere Priorität als der Sternoperator. Deshalb ist die Ausgabezeile folgendermaßen abzuändern:

```
1  cout << (*p).first_name << endl;
```

Es ist offensichtlich, dass dies keine übersichtliche Lösung ist. Andererseits wird diese Art des Zugriffes häufig benötigt. Deshalb gibt es dafür den Operator `->`, der genau das erledigt:

```
1  cout << p->first_name << endl;
```

5.1.2 Pointer und Konstanten

Im Zusammenhang mit Pointern wollen wir uns auch noch die Verwendung des Schlüsselwortes `const` ansehen. Wir kennen die Verwendung von `const` schon in der Verwendung als „normale“ Variablen, wie zum Beispiel in `const double pi{3.1415};`, wodurch festgelegt wird, dass die Variable `pi` nicht mehr verändert werden kann und daher als Konstante funktioniert.

Erstelle folgendes kleines Programm und versuche es zu übersetzen:

```
1  // const.cpp
2  #include <iostream>
3  #include <cmath>
4
5  using namespace std;
6
7  int main() {
8      const double pi{atan(1)*4};
9      const double e{exp(1)};
10
11     double* p;
12     p = &pi;
13 }
```

Das wird nicht funktionieren, da `p` ein Pointer auf ein `double` ist und wir hier `p` einen Pointer auf ein `const double` zuweisen wollen. Das lässt der Compiler jedoch nicht zu, da es sonst möglich wäre, über `p` den Wert von `pi` – zum Beispiel mittels `*p = 1;` – zu verändern. `pi` ist jedoch als `const` deklariert und darf

daher seinen Wert nicht ändern, womit klar ist, dass dies nicht erlaubt ist und vom Compiler zurückgewiesen wird.

Also müssen wir die Definition von `p` abändern:

```
1  const double* p;
```

Damit lässt sich das Programm auch problemlos übersetzen, da `p` ein Pointer auf ein `const double` und dieser Pointer mit der Adresse von `pi` initialisiert worden ist. Eine Änderung von `pi` über `p` ist daher nicht möglich.

Da der Pointer selber nicht konstant ist und wir in unserem Programm zwei Konstanten definiert haben, können wir auch `p` ändern:

```
1  p = &e;
```

Manchmal will man jedoch, dass der Pointer `p` selbst nicht veränderbar ist. Füge deshalb folgende Zeilen hinzu und versuche wieder zu übersetzen:

```
1  double d1{1};
2  double d2{2};
3  double* const q;
4  q = &d1;
```

Das wird sich nicht übersetzen lassen, da `q` ein konstanter Pointer ist und eine Konstante initialisiert werden muss.

So, damit unser Programm funktioniert, sind die letzten beiden Zeilen folgendermaßen abzuändern:

```
1  double* const q{&d1};
```

Tipp

Am besten liest man die Deklarationen von rechts nach links!

Eine Änderung von `q` auf `d2` ist damit natürlich nicht möglich, aber der Wert der Variable `d1` kann geändert werden:

```
1  *q = 2;
2  cout << d1 << endl;
```

Beide Varianten können auch kombiniert werden:

```
1  const double* const r{&e};
```

Noch eine letzte Bemerkung zu der Platzierung der Symbole `*` oder `&`: Leerraum wird auch hier ignoriert, damit könnte man auch `const double * const r{& e};` schreiben, aber das ist aus meiner Sicht nicht so gut zu lesen.

5.1.3 Probleme im Umgang mit rohen Pointern

Mehrere Probleme können im Zusammenhang mit Pointern auftreten:

- Das Verhalten von mehrmaligem Freigeben ist nicht definiert und führt in der Regel zu einem Programmabsturz. Du kannst es selber ausprobieren, indem du die Anweisung `delete p;` ein zweites Mal in dein Programm einfügst. Das Programm bricht mit einer Fehlermeldung ab!

  Fügst du allerdings vor dem zweiten `delete p;` eine Anweisung zum Setzen des Pointers auf `nullptr` ein: `p = nullptr`, dann wird das Programm nicht mehr abstürzen, da ein `delete` auf einen Nullzeiger keine Wirkung hat. Das bedeutet, dass es sinnvoll ist, nach dem Freigeben des Speichers den Pointer auf „Null“ zu setzen.

- Wird auf ein `delete` vergessen kommt es in der Regel zu einem Speicherleck (engl. *memory leak*), da das Speicherobjekt noch nicht zurückgegeben worden ist, aber auch nicht mehr darauf zugegriffen werden kann.

  Schreibe dazu folgende Anweisungen in dein Programm:

```
1  {
2      int* q{new int{}};
3  }
```

Es kommt zwar zu keiner zusätzlichen Ausgabe, aber es wird demonstriert, wie ein Speicherleck zustande kommen kann. Die Variable `q` ist eine lokale Variable in einem Block, da die Definition von dieser innerhalb von geschwungenen Klammern stehen. Der Speicher wird angefordert, aber nicht mehr freigegeben. Nach der geschlossenen geschwungenen Klammer steht die Variable `q` nicht mehr zur Verfügung, da sie den Geltungsbereich verlässt. Gleichzeitig wird `q` auch automatisch wieder gelöscht, jedoch nicht der Speicherbereich auf den `q` verweist. Damit kann auf diesen Speicherbereich nicht mehr zugegriffen werden.

Damit kommt es in der Regel bei lang laufenden Programmen wie z.B. bei Serveranwendungen zu Problemen, weil der verfügbare Speicher immer weniger wird.

- Es kann zu sogenannten hängenden Zeigern (engl. *dangling pointer*) kommen, das sind Zeiger, die auf ein nicht mehr gültiges Speicherobjekt verweisen. Das Verhalten ist in so einem Fall nicht definiert, sondern implementierungsabhängig.

  Schauen wir uns das an, indem wir folgenden Codeteil hinten an unser Programm anhängen:

```
1  p = new int{1};
2  int* r{p};
3  cout << "*r == " << *r << endl;
4  delete p;
5  cout << "*r == " << *r << endl;
6  *r = 2;
7  cout << "*r == " << *r << endl;
```

  Die Ausgabe sieht bei mir folgendermaßen aus:

```
1  *r == 1
2  *r == 0
3  *r == 2
```

  Es ist zu sehen, dass `r` auf den schon freigegebenen Speicherbereich verweist! Es gibt absolut keine Garantie, dass dieses Programm in irgendeiner Weise

funktioniert. Es kann sein, dass es kurzzeitig so aussieht, als ob es fehlerfrei ist, aber das ist ein schwerer Fehler, der entweder in weiterer Folge zum Absturz oder zu falschen Ergebnissen führt!

Wir sehen, dass es hier zu Problemen kommen kann. Eigentlich ist es so, dass es auf diese Art und Weise auch ganz sicher zu Problemen kommen wird! Daher biete ich einige Lösungen für diese Probleme an:

a. Einfach Pointer nicht verwenden: zumindest nicht auf diese Weise.
b. In C++11 gibt es in der Standardbibliothek sogenannte Smart-Pointer (eingedeutscht von *smart pointer*). Diese kümmern sich um die Speicherverwaltung. Wir werden diese im Abschnitt **5.4** auf der Seite **151** besprechen.
c. Ersatz der Pointer durch Referenzen, wenn dies möglich ist.
d. Muss oder will man Pointer unbedingt verwenden, dann sollte man auf das sogenannte RAII Prinzip zurückgreifen, dass später noch beschrieben wird.

5.2 Array

Ein Array (eingedeutscht von engl. *array*, deutsch *Feld*) hat einen bestimmten Typ und eine feste Größe. Es handelt sich bei diesem eingebauten Datentyp um ein low-level Werkzeug, das eigentlich nur zur Implementierung von Spezialdatenstrukturen verwendet werden sollte!

Arrays werden mit der folgenden Syntax definiert:

```
1 char firstname[10];
2 int results[100];
```

Bei der Variable `firstname` handelt es sich um ein Array von 10 `char` Elementen und bei `results` um ein Array von 100 `int` Elementen.

Initialisieren kann man Arrays ebenfalls mit der vereinheitlichten Initialisierung:

```
1 char firstname[]{'M', 'a', 'x', 0};
```

In diesem Fall ist eine Längenangabe innerhalb der eckigen Klammern nicht notwendig, da der Compiler die Anzahl der Elemente selbstständig bestimmt. Beachte in diesem Beispiel bitte, dass ein abschließendes Nullzeichen angegeben

wurde, da es sich sonst nicht um einen korrekten C-String handelt. Denke daran, dass dieses Nullzeichen das Ende eines C-Strings markiert.

Allerdings geht es für ein Array von `char` auch einfacher:

```
1 char firstname[]{"Max"};
```

Man kann auch die Längenangabe bei einem Array mitgeben, obwohl man eine Initialisierung vornimmt:

```
1 int coords[3]{1};
```

Hier wird die Längenangabe mitgegeben, aber nur ein Element initialisiert. Die restlichen Elemente werden mit Nullelementen initialisiert.

Hat man ein Array statisch angelegt wie gerade gezeigt, dann kann man über die Elemente eines solchen Arrays wie folgt iterieren:

```
1  // iterate_array.cpp
2
3  #include <iostream>
4
5  using namespace std;
6
7  int main() {
8      int coords[3]{1};
9
10     for (int i{0}; i < sizeof(coords) / sizeof(int); ++i) {
11         cout << coords[i] << endl;
12     }
13 }
```

Einfacher geht es natürlich mit der „foreach“-Variante von `for`:

```
cout << endl;

for (auto i : coords) {
    cout << i << endl;
}
```

Solche Arrays liegen am Stack. Will man Arrays am Heap dynamisch anfordern, dann ist zum Anlegen der Operator `new[]` und zum Freigeben der Operator `delete[]` zu verwenden:

```
// array.cpp
#include <iostream>

using namespace std;

int main() {
    int* points{new int[10]};

    for (int i{0}; i < 10; ++i) {
        points[i] = i;
    }

    for (int i{0}; i < 10; ++i) {
        cout << points[i] << ' ';
    }

    cout << endl;

    delete[] points;
}
```

Damit kann man allerdings nicht mehr die Bestimmung der Anzahl der Elemente mittels des `sizeof` Operators verwenden und auch die „foreach"-Variante von `for` funktioniert nicht mehr!

Dazu muss man Folgendes wissen: Ein Array wird von C++ prinzipiell wie ein Pointer auf das erste Element behandelt. Damit ergeben sich etliche Fallstricke, über die man ganz nett stolpern kann:

- Wird ein Array an eine Funktion übergeben, dann wird in Wirklichkeit nur der Pointer an die erste Stelle übergeben. Innerhalb der Funktion kennt C++ nur den Pointer! Wir haben dies schon bei der Funktion `main` bei den Kommandozeilenargumenten gesehen, dass zusätzlich die Anzahl der Arrayelemente übergeben wird.
- Eine Zuweisung von Arrays bewirkt nur die Zuweisung des einen Pointers auf den anderen Pointer! Das bedeutet, dass Arrays so nicht kopiert werden können.
- Vergleicht man zwei Arrays, dann werden nur die Pointer verglichen. Arrays können nur so verglichen werden, dass man selbst über alle Elemente iteriert und diese Elemente miteinander vergleicht.

Das bedeutet allerdings nicht, dass ein Array und ein Pointer das gleiche sind. Schauen wir uns das an dem folgenden Beispiel an:

```
1  char firstname[]{"Max"};
2  cout << firstname << endl;
3  char* lastname{"Mustermann"};
4  cout << lastname << endl;
```

Die Definition mittels eines Pointers wird der Compiler unter Umständen mit einer Warnung versehen, die besagt, dass es sich um eine veraltete Konvertierung handelt. Abgesehen davon werden diese Anweisungen soweit erwartungsgemäß funktionieren.

Hängt man allerdings die folgenden Zeilen an, dann wird der Unterschied deutlich:

```
1  firstname[0] = 'm';
2  lastname[0] = 'm';
```

Das Programm wird sich mit sich einem Zugriffsfehler beenden. Der Grund liegt daran, dass der Pointer `lastname` bei der Definition mit der Adresse auf den

Beginn des C-String-Literals initialisiert wurde. Wie wir später noch genauer betrachten werden, handelt es sich bei einem C-String-Literal um ein nicht veränderbares Array von Zeichen. Daher wäre es vernünftiger gewesen `lastname` folgendermaßen zu definieren:

```
1  const char* lastname{"Mustermann"};
```

Damit wäre einerseits die Warnung verschwunden und andererseits hätte der Compiler eine Fehlermeldung bei dem schreibenden Zugriff auf das Element mit dem Index 0 erzeugt (`lastname[0] = 'm'`).

Schauen wir uns in diesem Zusammenhang die sogenannte Zeigerarithmetik *kurz* an. Tausche in deinem Programm die gesamte zweite `for` Schleife durch folgende Schleife aus:

```
1  for (int i{0}; i < 10; ++i) {
2      cout << *(points + i) << ' ';
3  }
```

Was passiert hier?

- `points` wird also als Zeiger auf das erste Element betrachtet. Zu diesem Zeiger wird die ganze Zahl `i` hinzugezählt. Es werden aber zu der Adresse `points` nicht `i` Bytes hinzugezählt, sondern in unserem Fall `sizeof(int)` Bytes, da wir einen `int*` Pointer haben. Damit verweist `points + 2` auf die ganze Zahl mit dem Index 2.

- Danach wird mit dem Dereferenzierungsoperator `*` der Wert dieser Adresse genommen und ausgegeben. Damit funktioniert diese Schleife wie die vorhergehende Variante.

An sich kann man auch die Differenz von Pointern nehmen, wie z.B. `p2 - p1` und man bekommt die Anzahl der Elemente zurück, die zwischen diesen beiden Adressen liegen. Dazu müssen beide Pointer den gleichen Typ haben und `p2` größer als `p1` sein.

Auf alle Zeichen von `firstname` könnten wir auch folgendermaßen zugreifen:

```
1  for (char* p{firstname}; *p != 0; ++p)
2      cout << *p;
```

Hier nützen wir aus, dass ein C-String mit einem Nullzeichen abgeschlossen wird. Ansonsten funktioniert `++p` wie vorhin besprochen.

Als Benutzer greift man **immer** auf die Datentypen `std::vector` oder `std::array` zurück. Allerdings ist es wichtig das Konzept des Arrays verstanden zu haben, da viele Datentypen der Standardbibliothek mit Arrays arbeiten und Funktionen der Programmiersprache `C` nur mit solchen Arrays arbeiten. Solche Funktionen der Programmiersprache `C` können in `C++` ohne Probleme verwendet werden. Sie stehen in Headerdateien zur Verfügung, die in der Regel mit einem „c“ beginnen wie bei der Headerdatei `cmath`.

Eine offensichtliche Verwendung von Arrays haben wir schon bei der Verwendung von Kommandozeilenargumenten gesehen. Die Kommandozeilenparameter werden als Array von Pointern auf `char` an die Funktion `main` übergeben. Zusätzlich wird die Anzahl der Argumente als Parameter übergeben.

5.2.1 String-Literale

Dazu müssen wir uns außerdem ansehen welchen Typ C-String-Literale in `C++` haben. Der C-String-Literal `"C++"` hat den Typ `const char[4]`. Das bedeutet, dass es sich um ein Array mit 4 Elementen vom Typ `char` handelt, die nicht verändert werden können. 4 Elemente deshalb, weil zu den drei vorhandenen Zeichen noch das abschließende Nullzeichen `'\0'` als Endekennung angehängt wird. So eine Endekennung ist notwendig, da keine explizite Längeninformation gespeichert wird.

In einem String-Literal können die gleichen Escape-Sequenzen verwendet werden wie bei den Zeichenliteralen (im Abschnitt **4.4.1** beschrieben).

String-Literale, die nur durch Whitespace getrennt sind, werden vom Compiler aneinandergehängt. Betrachte folgendes Beispiel:

```
1  // stringliterals.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      const char* alphabet;
8      alphabet = "abcdefghijklmnopqrstuvwxyz\n"
9          "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
10     cout << alphabet << endl;
11 }
```

Dieses Beispiel zeigt auch, dass natürlich ein Zeilenumbruch direkt in ein C-String-Literal mittels `\n` eingefügt werden kann. Bei der Ausgabe wirkt sich dieser Zeilenumbruch natürlich entsprechend aus.

Weiters sieht man in diesem Beispiel, dass aufeinanderfolgende C-String-Literale vom Compiler zu einem C-String verkettet werden.

Trotzdem gibt es eine einfachere, elegantere Möglichkeit, nämlich die rohen C-String-Literale (engl. *raw character strings*). Diese sehen so aus, dass dem C-String-Literal der Präfix `R` vorangestellt wird und der eigentliche Inhalt der Zeichenkette in ein rundes Klammernpaar eingeschlossen wird. Zur Veranschaulichung probiere es aus, indem du den folgenden Quelltext hinten an das Programm anhängst:

```
1      alphabet = R"(abcdefghijklmnopqrstuvwxyz"\n
2  ABCDEFGHIJKLMNOPQRSTUVWXYZ)";
3      cout << endl << alphabet << endl;
```

Du siehst hier mehrere interessante Eigenheiten dieser rohen C-String-Literale:

- Es kommt ein doppeltes Hochkomma im String-Literal vor.
- Das String-Literal erstreckt sich über zwei Zeilen.
- Und es kommt das Escape-Zeichen im String vor, das keine Auswirkung hat.

Nicht schlecht, aber vielleicht stellst du dir jetzt die Frage, wie man in einen solchen String die Zeichenfolge `)"` packt, da diese ja das Ende des C-String-Literals kennzeichnet?

Dafür gibt es eine erweiterte Syntax, sodass man die Begrenzungszeichen verändern kann, indem man der offenen runden Klammer eine Zeichenkette voranstellt, die der schließenden runden Klammer hinten ebenfalls angehängt wird. Schauen wir uns das praktisch an, indem du die folgenden Zeilen an dein Programm anhängst:

```
1      alphabet = R"=(abcdefghijklmnopqrstuvwxyz)"\n
2  ABCDEFGHIJKLMNOPQRSTUVWXYZ)=";
3      cout << endl << alphabet << endl;
```

Es gibt allerdings auch noch andere Präfixe, die ähnlich den Zeichenliteralen funktionieren:

- `u8` bezeichnet einen UTF-8 String, wie z.B. `u8"Müßiggang"`, wobei davon ausgegangen wird, dass die Umlaute in UTF-8 kodiert sind. Alternativ kann auch UTF-8 kodierte Zeichen mittels Escape-Zeichen einfügen:

```
1  cout << u8"Müßiggang" << endl;
2  cout << u8"M\xc3\xbcßiggang" << endl;
```

- `u` bezeichnet einen UTF-16 String und funktioniert analog zu `u8`
- `U` bezeichnet einen UTF-32 String mit analoger Funktionsweise zu `u8` und `u`.
- `L` bezeichnet einen `wchar_t` string.

Jeder dieser Präfixe kann mit dem Präfix `R` kombiniert werden: `u8R`, `uR`, `UR` und `LR`.

5.2.2 Initialisierung

Wir haben schon die Faustregel kennengelernt, dass immer initialisiert werden soll. Als Ausnahme kann das Anlegen eines großen Speicherobjektes angesehen werden, das *zuerst* befüllt wird und erst *danach* wieder gelesen wird. Es macht daher keinen Sinn, das Speicherobjekt zuerst zu initialisieren und danach zu befüllen:

```
// large_buffer.cpp
#include <iostream>
#include <chrono>

using namespace std;

int main() {
    constexpr size_t max{100000};

    auto start = chrono::system_clock::now();

    for (int i{0}; i < max; ++i) {
        long long buffer[max];
    }

    auto end = chrono::system_clock::now();
    auto elapsed = chrono::duration_cast<chrono::milliseconds>(end - start);
    cout << elapsed.count() << "ms" << endl;
}
```

Als Ausgabe erhalte ich:

```
0ms
```

Hier verwende ich Möglichkeiten der Zeitmessung aus der Standardbibliothek. Wir werden uns dies in weiterer Folge später ansehen.

Ändere jetzt den Quelltext so um, dass der Puffer *initialisiert* wird:

```
long long buffer[max]{};
```

Die Ausgabe auf meinem Rechner sieht zum Beispiel so aus:

```
2593ms
```

Man sieht, dass die zeitliche Differenz in Spezialfällen durchaus beachtlich ausfallen kann!

5.3 Referenz

Wie schon erwähnt, ist eine Referenz nichts anderes als ein anderer Namen für eine Speicherstelle.

Schauen wir uns dazu noch einmal ein einfaches Beispiel an:

```
1  // lreference.cpp
2  using namespace std;
3
4  int main() {
5      int x{1};
6      int& r{x};
7
8      r = 2;
9      cout << "x = " << x << endl;
10 }
```

Da es sich bei `r` um eine Referenz also einen anderen Namen für die Speicherstelle von `x` handelt, wird bei einer Veränderung mittels `r` in Wirklichkeit der Wert von `x` verändert.

Wir erkennen also, dass wir das Beispiel auch mit Hilfe von Zeigern programmieren hätten können:

```
1  int* p{nullptr};
2  p = &x;
3  *p = 3;
4  cout << "x = " << x << endl;
```

Es gibt allerdings ein paar nicht unbedeutende Unterschiede zwischen der Verwendung von Zeigern und Referenzen:

- Offensichtlich ist die Syntax verschieden: `r = 2;` im Vergleich zu `*p = 3`!

Hier sieht man schön, dass die Syntax eines Pointers anders ist und auch, dass Pointer umständlicher zu verwenden sind.

- Ein Pointer kann zu verschiedenen Objekten zu verschiedenen Zeitpunkten zeigen, während eine Referenz bei der Definition initialisiert werden muss und in weiterer Folge nicht mehr verändert werden kann.

  Besteht also die Notwendigkeit, dass zu verschiedenen Zeiten auf verschiedene Objekte zugegriffen werden muss, dann muss man sich für Pointer entscheiden.

- Ein Pointer kann den Nullwert haben, während eine Referenz immer initialisiert werden muss. Damit muss man sich bei einer Referenz keine Gedanken machen, ob diese initialisiert wurde oder nicht.

  Aus diesem Gesichtspunkt betrachtet ist es sicherer, eine Referenz zu verwenden.

- Ein Zugriff über eine Referenz kann in bestimmten Fällen ohne Indirektion erfolgen, da die Referenz nichts anderes als ein anderer Name ist.

  Die Indirektion über einen Pointer funktioniert folgendermaßen: Bei einem Pointer muss zuerst auf den Wert des Pointers zugegriffen werden, dann muss dieser Wert als Adresse interpretiert werden und danach kann auf die Speicherstelle zugegriffen werden, die diese Adresse hat und der Wert der Speicherstelle ausgelesen oder verändert werden.

  Ist die Performance das vorrangige Ziel, dann spricht das für die Verwendung von Referenzen.

  Bei Verwendung von Referenzen respektive Pointern bei der Parameterübergabe an Funktionen gibt es diesbezüglich keinen Unterschied!

- Da es sich bei einer Referenz nicht um ein Objekt im Sinne einer „normalen“ Variablendefinition handelt, kann man keinen Pointer auf eine Referenz haben und auch kein Array von Referenzen anlegen!

  Ist daher der Bedarf gegeben, dass man den Zugriff über eine doppelte Indirektion herstellen muss oder muss man ein Array verwenden, dann ist die Verwendung von Pointern obligatorisch.

Es gibt prinzipiell zwei Arten von Referenzen:

- lvalue Referenz (engl. *lvalue reference*) ist eine Referenz auf einen lvalue. Das haben wir uns gerade angesehen.

- rvalue Referenz (engl. *rvalue reference*) ist eben eine Referenz auf einen rvalue.

5.3.1 lvalue Referenz

Eine lvalue Referenz kann entweder ohne `const` oder mit `const` auftreten.

Wird die lvalue Referenz ohne `const` verwendet, d.h. so wie wir das bis jetzt betrachtet haben, dann muss diese Referenz mit einem lvalue des entsprechenden Typs initialisiert werden. Das bedeutet, dass der Compiler die folgende Deklaration beanstanden wird:

```
1 int* q{nullptr};
2 int& r1{0};
3 int& r2{q};
```

Bei `r1` wird nicht mit einem lvalue initialisiert und bei `r2` stimmt der Typ der Referenz von `r2` (`int`) nicht mit dem Typ des lvalue (`int*`) überein!

Bei einer lvalue Referenz mit `const` muss der Initialisierer nicht ein lvalue sein und nicht einmal mit dem Typ übereinstimmen. In solchen Fällen wird vom Compiler folgendermaßen vorgegangen:

a. Zuerst wird eine implizite Konvertierung durchgeführt, sodass die Typen übereinstimmen.
b. Dann wird der erhaltene Wert in einer temporären Variable abgelegt.
c. Diese temporäre Variable wird zur Initialisierung der Referenz verwendet. Die Lebenszeit solch einer temporären Variable endet, wenn die Referenz den Geltungsbereich verlässt.

Schauen wir uns dazu die folgenden Codezeilen an:

```
1 {
2     const int& r{1};
3     cout << r << endl;
4 }
```

Die temporäre Variable, die intern angelegt wird, wird bei der geschlossenen geschwungenen Klammer des Blockes wieder entfernt.

lvalue Referenzen werden oft im Zuge von „foreach“ Schleifen verwendet. Hänge den folgenden Code hinten an die Datei an:

```
1  string long_names[]{"maxi", "mini", "otto"};
2
3  for (const auto& name : long_names) {
4      cout << name << endl;
5  }
```

- Zuerst wird ein Array von `string` Objekt angelegt, wobei jedes einzelne `string` Objekt aus einem C-String-Literal erzeugt wird.
- Der interessante Teil betrifft die Laufvariable. Prinzipiell teilen wir dem Compiler mit, dass er den Typ selbst eruieren soll, aber wir wollen den Wert der Laufvariable nicht ändern (`const`) und außerdem wollen wir, dass eine lvalue Referenz verwendet wird.

  Das bewirkt, dass die Strings nicht kopiert werden, sondern lediglich das `string` Objekt aus dem Array direkt verwendet wird. Bei wirklich langen Namen erspart man sich sehr viel an Speicher, der zuerst angefordert, dann kopiert und am Ende des Schleifendurchganges wieder freigegeben werden muss. Das wird sich in der Regel auch in der Laufzeit entsprechend auswirken.

5.3.2 rvalue Referenz

Eine rvalue Referenz referenziert einen rvalue, der entsteht, wenn ein temporäres Objekt im Zuge der Auswertung eines Ausdrucks entsteht. Auch eine rvalue Referenz muss bei der Definition initialisiert werden, nämlich mit einem rvalue!

Schauen wir uns dazu einmal folgendes Programm an, das vorerst ohne Referenz auskommt:

```
1  // rreference.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  string f() {
7      return "f()";
8  }
9
10 int main() {
11     string res{f()};
12     cout << res << endl;
13 }
```

Das Interessante ist, welche Objekte beim Aufruf der Funktion `f` erzeugt werden:

a. Zuerst wird bei der `return`-Anweisung aus dem C-String-Literal ein temporäres Objekt vom Typ `string` angelegt.

b. Dann wird dieses Objekt an den Aufrufer zurückgegeben. Dazu muss eine Kopie erstellt werden, die in der Funktion `main` auf der rechten Seite der Zuweisung `res = f()` zur Verfügung steht.

c. Beim Ausführen der Initialisierung muss das zurückgegebene temporäre Objekt wiederum in das Zielobjekt `res` kopiert werden.

Es steht dem Compiler frei mittels Optimierungen temporäre Objekte zu vermeiden, aber ohne temporäre Objekte geht es in diesem Fall nicht. Besonders den letzten Schritt wird ein Compiler wegoptimieren, wodurch faktisch nur mehr zwei Kopieraktionen über bleiben. Wir werden uns das im Zuge der Behandlung des Kopierkonstruktors im Abschnitt **11.2** noch genauer ansehen.

Ändere jetzt den Typ von `res` auf `string&` um:

```
1  string& res{f()};
```

Der Compiler wird das Programm nicht mehr übersetzen, da es sich bei dem Wert des Ausdruckes von `f()` nicht um einen lvalue sondern um eine lvalue-Referenz handelt!

Ändere das Programm so ab, dass eine rvalue-Referenz als Typ von `res` verwendet, also `string&&` wird:

```
1  string&& res{f()};
```

Das Programm funktioniert wieder wie zuvor! Warum also rvalue-Referenzen verwenden und nicht einfach `string res{f()};`? Wir haben schon gesehen, dass bis zu drei Mal kopiert werden muss. Handelt es sich um ein großes Objekt, dann können diese Kopieraktionen durchaus ins Gewicht fallen. Verwendet man eine rvalue-Referenz, dann kann der Compiler in diesem Fall eine Kopieraktion durch eine Verschiebeaktion austauschen. Bei einer Verschiebeaktion wird der Inhalt des Objektes nicht kopiert, sondern in das neue Objekt verschoben. Wir werden uns das noch genauer im Abschnitt **11.2** ansehen.

Damit erkennen wir, dass es für C++ Sinn macht sowohl lvalue-Referenzen als auch rvalue-Referenzen im Repertoire zu haben:

- Eine nicht-konstante lvalue-Referenz bezieht sich auf ein Objekt, das vom Benutzer beschrieben werden kann.
- Eine konstante lvalue-Referenz verweist auf eine Konstante, die aus Sicht des Benutzers der Referenz nicht veränderbar ist.
- Eine rvalue-Referenz bezieht sich auf ein temporäres Objekt, das verändert werden kann, da es nicht mehr benützt wird. Ein temporäres Objekt kann nicht mehr benützt werden, da es keine Möglichkeit gibt auf dieses Objekt in weiterer Folge zuzugreifen. Damit können die Daten, die in diesem Objekt gespeichert sind, verschoben werden und müssen nicht kopiert werden.

Verwenden wir explizit rvalue-Referenzen, dann kann der Compiler Verschiebeaktionen durchführen, wenn diese möglich sind. Manchmal ist es jedoch so, dass wir wissen, dass ein Verschieben möglich ist, der Compiler dies jedoch nicht erkennen kann. In so einem Fall müssen wir dem Compiler die entsprechenden Informationen liefern.

Nehmen wir einmal an, dass wir eine ganz „normale“ Funktion zum Vertauschen der Argumente schreiben wollen. Solch eine Funktion wird oft „swap“ genannt.

```
1  #include <iostream>
2
3  using namespace std;
4
5  void swap(string& a, string& b) {
6      string tmp{a};
7      a = b;
8      b = tmp;
9  }
10
11 int main() {
12     string s1{"foo"};
13     string s2{"bar"};
14
15     cout << "s1 = " << s1 << endl;
16     cout << "s2 = " << s2 << endl;
17     swap(s1, s2);
18     cout << "s1 = " << s1 << endl;
19     cout << "s2 = " << s2 << endl;
20 }
```

Die Ausgabe schaut wie erwartet folgendermaßen aus:

```
1  s1 = foo
2  s2 = bar
3  s1 = bar
4  s2 = foo
```

Die Funktion `swap` bekommt ihre beiden Parameter als Referenz, womit auch in gewisser Weise die Funktion zwei Werte zurückgeben kann. Der eigentliche Rückgabewert wurde hier nicht verwendet. Damit einhergehend kommen auch keine Kopieraktionen bei der Übergabe der beiden Parameter zustande.

Das klingt gut, aber leider kommt es sehr wohl zu Kopieraktionen, da bei der Initialisierung der Variable `tmp` die erste Kopieraktion durchgeführt wird, bei der Zuweisung an `a` und danach an `b` die nächsten beiden Kopieraktionen!

Wenn man sich überlegt, dass wir eigentlich die Variablen nicht kopieren wollen, sondern nur deren Inhalte verschieben, dann müssen wir unsere Funktion auf die Verwendung von rvalue-Referenzen umstellen:

```
1  void swap(string& a, string& b) {
2      string tmp{static_cast<string&&>(a)};
3      a = static_cast<string&&>(b);
4      b = static_cast<string&&>(tmp);
5  }
```

Diese Umstellung bewirkt, dass nicht mehr die Inhalte der Strings kopiert werden, sondern die Inhalte verschoben werden:

- Nach der Initialisierung der Variable `tmp` hat die Variable `a` keinen Inhalt mehr, da dieser zur Variable `tmp` verschoben wurde.

  Das wird vom Compiler deshalb durchgeführt, da die Variable `a` explizit zu einer rvalue-Referenz konvertiert wird. Damit kann der Compiler eine spezielle Operation zur Initialisierung von `tmp` aufrufen, die die Klasse `string` zur Verfügung stellt und die Verschiebe-Operation durchführt.

  Wie diese spezielle Verschiebe-Operation genau aussieht und wie man eine solche selber programmieren kann, werden wir uns im Abschnitt **11.2** noch detailliert ansehen.

- Bei den Zuweisungen zu den Variablen `a` und `b` funktioniert dies ähnlich, nur dass diese Variablen nicht initialisiert werden, sondern zugewiesen. Aufgrund der Konvertierung zu einer rvalue-Referenz kann der Compiler eine spezielle Operation zur Zuweisung aufrufen, die die Klasse `string` ebenfalls zur Verfügung stellt und wiederum die Verschiebe-Operation durchführt.

Damit kommt diese Funktion ohne aufwändige Kopieroperationen aus! Das ist nun einmal nicht so schlecht, allerdings ist die Notation nicht gerade einfach und eingängig. Deshalb stellt die Standardbibliothek eine Funktion `move` im Header `<utility>` (wird schon von `<iostream>` inkludiert) zur Verfügung, die nichts anderes macht, als die Konvertierung in eine rvalue-Referenz vorzunehmen:

```
1  void swap(string& a, string& b) {
2      string tmp{move(a)};
3      a = move(b);
4      b = move(tmp);
5  }
```

Damit schaut die Funktion schon viel übersichtlicher aus und bewirkt genau das Gleiche! Vielleicht könnte man meinen, dass damit die Performance leidet, da zusätzliche Funktionen aufgerufen werden, aber dem ist nicht so! Die Funktionen werden vom Compiler aufgerufen und nicht zur Laufzeit. Solche Funktionen werden wir uns detailliert im Abschnitt **13.2** ansehen.

Aber es geht eigentlich noch viel einfacher, weil eine solche Funktion `swap` schon im Header `utility` vorhanden ist, die mit prinzipiell *allen* Typen umgehen kann. Auch so etwas werden wir uns im Abschnitt **13.2** noch genauer ansehen.

Damit du die eingebaute Funktion `swap` verwenden kannst, brauchst du nur die Definition der selbst geschriebenen Funktion `swap` aus deinem Programm löschen!

5.4 Smart-Pointer

Smart-Pointer stellen eine Möglichkeit dar, mit den besprochenen Problemen beim Umgang mit rohen Pointer, zu handhaben. Diese Smart-Pointer haben den Zweck, sich um die Speicherverwaltung zu kümmern und sicherzustellen, dass der Speicher genau zum richtigen Zeitpunkt wieder freigegeben wird.

Smart-Pointer stellen also eine Abstraktionsschicht zur Verfügung, die die Freigabe des Speichers in die Implementierung des Smart-Pointers verlagert. Das Prinzip ist, dass der Speicher freigegeben wird, wenn die Variable des Smart-Pointers den Block verlässt, in dem diese definiert wird.

Die Standardbibliothek von C++11 stellt verschiedene Arten von Smart-Pointer zur Verfügung, die wir uns in Folge ansehen werden.

5.4.1 `unique_ptr`

Zuerst wollen wir uns dem sogenannten „eindeutigen Smart-Pointer“ `unique_ptr` (engl. *unique pointer*) zuwenden, den wir ab jetzt einfach mit Unique-Pointer bezeichnen wollen.

Dieses „unique“ bedeutet, dass dieser Smart-Pointer für das Speicherobjekt auf das dieser verweist, die Verantwortung übernimmt. Der Smart-Pointer „besitzt“ das Speicherobjekt. Das bedeutet, dass dieser Smart-Pointer die Verantwortung über die korrekte Entfernung des Speicherobjektes übernimmt. Weiters gibt es exakt eine Kopie von dieser Art von Smart-Pointer zu einer Zeit. Was dies genau bedeutet werden wir uns jetzt ansehen.

Schauen wir uns dazu vorerst das einfachste Beispiel an und gehen die verschiedenen Aspekte nach und nach durch.

Zuerst wollen wir den Fall durchspielen, dass vergessen wurde den Speicher freizugeben. Unter Verwendung von `unique_ptr` kann dies nicht mehr passieren:

```
1  // smartpointer.cpp
2  #include <iostream>
3  #include <memory>
4
5  using namespace std;
6
7  int main() {
8      {
9          unique_ptr<int> pi{new int{1}};
10     }
11 }
```

Es ist klar, dass dieses Programm keine besondere Funktionalität hat, trotzdem folgt hier eine Erklärung der Funktionsweise:

- Zuerst wird die Headerdatei `memory` eingebunden, die die verschiedenen Arten der Smart-Pointer zur Verfügung stellt.
- Dann wird innerhalb von `main` ein Block erzeugt, der uns zeigen soll, dass der Speicher wieder freigegeben wird.
- Innerhalb dieses Blockes wird ein `unique_ptr` angelegt, der auf einen `int` verweisen kann. Zusätzlich wird dieser mit einem Pointer auf ein neu am Heap angelegtes `int`-Objekt initialisiert. Damit übernimmt dieser Smart-Pointer die Kontrolle über das Speicherobjekt.
- Beim Verlassen des Blockes endet die Lebenszeit des Smart-Pointers und es wird dieser entfernt. Beim Entfernen des Smart-Pointers wird eine spezielle

Funktion (der Destruktor, siehe später) aufgerufen, der den Speicher des Speicherobjektes wieder freigibt. Damit gibt es keine Möglichkeit mehr, das Freigeben des Speichers zu „vergessen“.

Natürlich kann man argumentieren, dass man „eh nicht vergisst“. Ja, aber schwieriger wird es, wenn Pointer an Funktionen übergeben werden oder von Funktionen zurückgeliefert werden. Es müsste nämlich geklärt sein, wer für sich für das Speicherobjekt verantwortlich fühlt und sich letztendlich um die Freigabe des Speichers kümmert.

Bevor wir uns allerdings noch genauer mit den verschiedenen Möglichkeiten des Einsatzes von Smart-Pointer auseinandersetzen, wollen wir uns vorerst die Frage stellen, wie man eigentlich auf das referenzierte Speicherobjekt zugreift. Die Antwort ist einfach: Wie bei einem normalen Pointer!

Füge einfach die nachfolgende Ausgabezeile direkt hinter der Definition des `unique_ptr` an und probiere es aus:

```
1  cout << *pi << endl;
```

Wir sehen, dass der Operator `*` offensichtlich auch für Smart-Pointer überladen wurde, damit diese sich wie echte Pointer verhalten.

Wer ist nun für die Freigabe eines Speicherobjektes am Heap verantwortlich? Schauen wir uns vorerst dazu die folgende Situation für normale Pointer an:

```
1  void use_ptr(int* pi) {
2      cout << *pi << endl;
3  }
4
5  int* pi{new int{2}};
6  use_ptr(pi);
```

Die Frage ist, wer sich um die Freigabe des Speichers kümmern soll? Die Funktion `use_ptr` oder der Aufrufer? Was wenn beide den Speicher wieder freigeben oder wenn keiner den Speicher freigibt. Was passiert, wenn die Funktion den Speicher freigibt, aber der Aufrufer noch auf den Speicher zugreifen will?

Weiters könnte es sein, dass die Funktion `use_ptr` eine Kopie des übergebenen Pointers anlegt und diesen später weiterverwendet? Das betrifft hauptsächlich die

Situation, wenn es sich bei der Funktion um eine Methode (engl. *method*, in C++ meist *member function* genannt) handelt.

Damit kommen wir zur eigentlichen Fragestellung, nämlich, wer die Eigentümerschaft über den Pointer erhält: Der Aufrufer oder die aufgerufene Funktion? Die Antwort kann natürlich je nach Anwendungsfall unterschiedlich sein. Es muss lediglich klar sein, wer die Verantwortung hat und derjenige Teil muss sich um die Freigabe kümmern. Und nach einer Freigabe darf der andere Teil nicht mehr auf das Speicherobjekt zugreifen.

Auch hier kann man `unique_ptr` einsetzen!

Schaue dir dazu die folgende naheliegende Lösung an:

```
1  // unique_ptr.cpp
2  #include <iostream>
3  #include <memory>
4
5  using namespace std;
6
7  void use_ptr(unique_ptr<int> upi) {
8      cout << *upi << endl;
9  }
10
11
12 int main() {
13     unique_ptr<int> upi{new int{1}};
14     use_ptr(upi);
15 }
```

Diese Lösung lässt sich jedoch, aus gutem Grund, nicht übersetzen. Das liegt daran, dass man `unique_ptr` nicht kopieren kann. Könnte man Unique-Pointer kopieren, dann wäre in diesem Fall kein Unterschied zu einem normalen Pointer gegeben! Und damit einhergehend wäre wiederum nicht klar, wer die Verantwortung für diesen Pointer hätte.

Eine einfache Lösung ist, keine Kopie des Smart-Pointers übergeben zu wollen, sondern eine Referenz. Wird eine Referenz übergeben, dann bedeutet das, dass die Adresse des Smart-Pointers an die Funktion übergeben wird. Damit lässt sich das Beispiel jetzt übersetzen:

```
1  void use_ptr(unique_ptr<int>& upi) {
```

Die Bedeutung dieser Änderung ist, dass der Aufrufer weiterhin die Verantwortung über den Smart-Pointer behält.

Was jedoch, wenn der Aufrufer die Kontrolle über den Smart-Pointer an die Funktion übergeben will? In diesem Fall muss man folgendermaßen vorgehen:

```
1  void use_ptr(unique_ptr<int> upi) {
2      cout << *upi << endl;
3  }
4
5
6  int main() {
7      unique_ptr<int> upi{new int{1}};
8      use_ptr(move(upi));
9      cout << ((upi.get() == nullptr) ? 0 : *upi) << endl;
10 }
```

Was passiert?

- Zuerst sehen wir, dass die Funktion den eigentlichen Unique-Pointer wieder ohne Referenz erhält. Das bedeutet, dass eine Kopie am Stack übergeben wird. Allerdings haben wir gesehen, dass man einen Unique-Pointer nicht als Kopie übergeben kann.

- Im Aufruf der Funktion sehen wir, dass der Unique-Pointer jetzt in einen Aufruf der Funktion `move()` gekapselt ist. Damit wird aus dem lvalue `upi` vom Typ `unique_ptr<int>` ein rvalue vom Typ `unique_ptr<int>&&`. Damit kann der Inhalt, also der rohe Pointer, in den Parameter `upi` der Funktion `use_ptr` verschoben werden.

- In der Funktion `use_ptr` ist das Argument `upi` jetzt im Besitz des verschobenen rohen Pointers. Am Ende der Funktion endet auch die Lebenszeit des Argumentes `upi` und damit wird auch der Speicher freigegeben auf den der rohe Pointer verweist.

- Die Ausgabe wird folgendermaßen aussehen:

```
1
2
```

```
1
0
```

Das liegt eben daran, dass der rohe Pointer in den Parameter der Funktion verschoben worden ist. Bei dieser Verschiebe-Operation wurde der alte rohe Pointer auf den Wert `nullptr` gesetzt. Mittels `get` erhält man den rohen Pointer.

Der Typ `unique_ptr` kennt noch weitere Methoden und überladene Methoden.

Leider kann es noch zu folgendem Problem kommen, dass Exceptions auftreten und Speicher nicht freigegeben wird, obwohl man Unique-Pointer verwendet:

```
1
2
3
4
5
```

```
void use_ptr(unique_ptr<Game> a, unique_ptr<Game> b);

// Verwendung:
use_ptr(unique_ptr<Game>{new Game{1}},
        unique_ptr<Game>{new Game{2}});
```

Bei Game soll es sich um einen benutzerdefinierten Datentyp handeln, der über einen Konstruktor verfügt. Wir gehen davon aus, dass dieser Konstruktor unter Umständen auch eine Exception werfen kann.

Das Problem liegt wiederum darin, dass der Compiler die Ausdrücke auswerten darf wie er möchte. Das bedeutet, dass es zu folgender Auswertungsreihenfolge kommen könnte:

a. Speicher für das erste `Game`-Objekt anfordern
b. Konstruktor für `Game{1}` ausführen
c. Speicher für das zweite `Game`-Objekt anfordern
d. Konstruktor für `Game{2}` ausführen
e. `unique_ptr<Game>` für erstes `Game`-Objekt anlegen
f. `unique_ptr<Game>` für zweites `Game`-Objekt anlegen
g. `use_ptr` aufrufen

Jetzt könnte es sein, dass zum Beispiel beim Anfordern des Speichers für das zweite `Game`-Objekt eine Exception geworfen wird oder der Konstruktor `Game{2}` eine Exception wirft. Dann ist ein Speicherleck entstanden, da das erste `Game`-Objekt nicht mehr freigegeben worden ist.

Es gibt prinzipiell zwei Lösungen:

- Man zerlegt diesen Aufruf der Funktion `use_ptr` in mehrere Anweisungen:

```
1 unique_ptr<Game> game1{1};
2 unique_ptr<Game> game2{2};
3 use_ptr(move(game1), move(game2));
```

- Ab C++14 gibt es noch die Funktion `make_unique`, die folgendermaßen verwendet werden kann und diese Probleme nicht aufweist:

```
1 use_ptr(make_unique<Game>(1), make_unique<Game>(2));
```

  Steht noch kein C++14 Compiler zur Verfügung, dann kann man diese Funktion als Template leicht selber definieren:

```
1 template<typename T, typename... Args>
2 std::unique_ptr<T> make_unique(Args&&... args) {
3     return {new T(std::forward<Args>(args)...)};
4 }
```

5.4.2 `shared_ptr`

Komplizierter wird die Situation, wenn man die Verantwortung über den Speicher aufteilen muss, da keinem Objekt die klare Verantwortung zugewiesen werden kann. Das bedeutet, dass der Besitz eines Speichers aufgeteilt wird. Aus genau diesem Grund wird so ein Smart-Pointer als Shared-Pointer bezeichnet.

Angelegt werden kann ein Shared-Pointer wie ein Unique-Pointer:

```
1  // shared_ptr.cpp
2  #include <iostream>
3  #include <memory>
4
5  int main() {
6      shared_ptr<int> spi{new int{1}};
7  }
```

Die Unterschiede werden allerdings erst dann sichtbar, wenn Shared-Pointer kopiert werden:

```
1  cout << spi.use_count() << endl;
2  {
3      shared_ptr<int> spi2{spi};
4      cout << spi2.use_count() << endl;
5  }
6  cout << spi.use_count() << endl;
```

Die Ausgabe wird folgendermaßen aussehen:

```
1  1
2  2
3  1
```

Was bedeutet diese Ausgabe und was passiert in dem Programm?

- Die erste Ausgabe bedeutet, dass es einen Akteur gibt, der auf die ganze Zahl 1 zugreifen kann, die am Heap liegt. Anders ausgedrückt: Es gibt genau einen Besitzer dieses Speicherobjektes.
- Danach wird innerhalb eines Blockes eine weitere lokale Shared-Pointer Variable `spi2` angelegt und mit einer Kopie von `spi` initialisiert. Du siehst, dass man Shared-Pointer sehr wohl kopieren kann.
- Die nachfolgende Ausgabe von 2 zeigt uns an, dass es jetzt 2 Besitzer dieses Speicherobjektes gibt. Diese Ausgabe zeigt uns, dass mit dem Initialisieren von

spi2 durch eine Kopie von spi erkannt worden ist, dass die Anzahl der Besitzer um eins erhöht hat.

- Die letzte Ausgabe zeigt uns, dass es nur mehr einen Besitzer gibt. Das bedeutet, dass es mit dem Entfernen der lokalen Variable spi2 auch einen Besitzer weniger gibt.

Analog zu Unique-Pointern ist es so, dass das Speicherobjekt freigegeben wird, wenn es keine Besitzer mehr gibt!

Zusätzlich zum Kopieren von Shared-Pointern ist es auch möglich diese zu verschieben, wie wir es schon von den Unique-Pointern kennen. Weiters gibt es, wie bei Unique-Pointer, die gewohnten überladenen Operatoren und auch weitere Methoden.

Ab C++11 gibt es, analog zu make_unique in C++14, eine Funktion make_shared, die ähnliche Funktionalität hat:

```
1  auto point = make_shared<double>(2.5);
```

Zusätzlich zu dem Vorteil, dass die Verwendung dieser Funktion für sicheres Verhalten auch im Falle von Exceptions sorgt, ist diese auch von der Performance dem expliziten Anlegen mittels shared_ptr<double>{new double{2.5}} vorzuziehen.

Wenn wir uns die Implementierung benutzerdefinierter Datentypen wie Klassen ansehen, dann bekommen die Shared-Pointer mehr an Bedeutung.

Wie wird jedoch diese Funktionalität implementiert? Im Standard ist keine Implementierung vorgeschrieben, aber die gängigen Implementierungen verwenden einen Referenzzähler, der die Anzahl der Referenzen zählt.

Es kann ein Problem im Zusammenhang mit Shared-Pointern geben, nämlich, wenn es einen Zyklus in der Verwendung gibt. Schauen wir uns dazu folgendes Beispiel an:

```
1  // shared_ptr_cycle.cpp
2  struct Engine;
3
4  struct Ship {
5      shared_ptr<Engine> engine;
6      ~Ship() {
7          cout << "Ship destructed" << endl;
8      }
9  };
10
11 struct Engine {
12     shared_ptr<Ship> ship;
13     ~Engine() {
14         cout << "Engine destructed" << endl;
15     }
16 };
17
18 int main() {
19     {
20         shared_ptr<Ship> ship{new Ship};
21         shared_ptr<Engine> engine{new Engine};
22         cout << ship.use_count() << ", "
23              << engine.use_count() << endl;
24         ship->engine = engine;
25         engine->ship = ship;
26         cout << ship.use_count() << ", "
27              << engine.use_count() << endl;
28     }
29     Ship ship2;
30 }
```

Wir finden hier eine neue syntaktische Notation: Man kann innerhalb eines `struct` auch Funktionen anschreiben, die in diesem Fall Methoden genannt werden (siehe Abschnitt **8.2**). Eine spezielle Methode ist der Destruktor, den wir erstmals im Abschnitt **3.8.1** kennengelernt haben und innerhalb einer Klasse (hier `struct`) mittels einer Tilde `~` gekennzeichnet ist. Dieser Destruktor wird eben

aufgerufen, wenn das Objekt entfernt wird und ist dafür da, noch benutzerdefinierte Aufräumaktionen durchzuführen. Genauer werden wir uns dies noch im Abschnitt 8.4 ansehen.

Nach dem Verlassen des Blockes werden wir ein Speicherleck haben, das auch sehr schön in der Ausgabe zu erkennen ist:

```
1, 1
2, 2
Ship destructed
```

Diese Ausgabe kommt alleinig von der Instanzvariable `ship2`! Wie kommt es jetzt dazu, dass weder `ship` noch `engine` entfernt werden?

Nach dem Anlegen der beiden Shared-Pointer `ship` und `engine` stehen die Referenzzähler jeweils auf 1. Nach den beiden Zuweisungen haben die Referenzzähler beide den Wert 2. Beim Verlassen des Blockes werden die beiden Shared-Pointer entfernt, aber die referenzierten Objekte vom Typ `Ship` und `Engine` verweisen noch immer über Shared-Pointer aufeinander. Also werden die Referenzzähler noch beide den Wert 1 aufweisen und damit werden auch die beiden Objekte vom Heap nicht gelöscht werden!

Auf diese beiden Objekt kann jedoch nicht mehr zugegriffen werden und damit erkennen wir, dass es zu einem Speicherleck gekommen ist. Schuld daran ist der Zyklus über die beiden Shared-Pointer!

Was kann dagegen getan werden? Der Zyklus muss entfernt werden! Wir werden die Klasse `Engine` wie folgt umbauen, indem wir den `shared_ptr` in einen `weak_ptr` umändern:

```
struct Engine {
    weak_ptr<Ship> ship;
    ~Engine() {
        cout << "Engine destructed" << endl;
    }
};
```

Jetzt sieht die Ausgabe dazu folgendermaßen aus:

```
1, 1
1, 2
Ship destructed
Engine destructed
Ship destructed
```

Hier ist sehr schön, die Auswirkung des Weak-Pointers zu erkennen: Dieser bewirkt keine Erhöhung des Referenzzählers der `Ship`-Instanz! Das bedeutet, dass dieser nur ein „schwacher“ Pointer (engl. *weak pointer*) ist.

Damit ergibt sich allerdings auch, dass man sich bei einem Weak-Pointer nicht sicher sein kann, ob dieser noch auf ein Speicherobjekt verweist oder nicht:

```
#include <memory>
#include <iostream>

using namespace std;

int main() {
    weak_ptr<int> wpi;
    {
        auto spi = make_shared<int>(1);
        wpi = spi;
    }
    cout << *wpi << endl;
    }
```

Das wird *nicht* übersetzen, da `weak_ptr` keinen überladenen Operator `*` hat! Das macht auch gar keinen Sinn, da nicht sicher ist, dass das referenzierte Objekt noch existiert.

Um auf das Speicherobjekt über einen Weak-Pointer zuzugreifen, muss man sich sicher sein, dass dieses noch existiert. Dafür bietet `weak_ptr` die Methode `lock` an, die einen Shared-Pointer zurückliefert, der auf das Speicherobjekt referenziert, wenn es eines gibt, anderenfalls wird ein „Null“-Shared-Pointer zurückgeliefert.

Tauschen wir jetzt bei unserem Beispiel den falschen Ausdruck `*wpi` durch `*(wpi.lock())` aus. Es wird jetzt übersetzen, aber das Programm wird vermutlich abstürzen! Das liegt daran, dass zu diesem Zeitpunkt das Speicherobjekt nicht mehr existiert, da `spi` nicht mehr existiert. Damit liefert `lock` einen Shared-Pointer zurück, der einen Null-Shared-Pointer entspricht. Ändere daher die Ausgabe so ab, dass diese folgendermaßen aussieht:

```
1  cout << wpi.lock().get() << endl;
```

Die Methode `get()` eines `shared_ptr` liefert den rohen Pointer auf das Speicherobjekt zurück. Die Ausgabe davon wird `0` sein, ein Null-Pointer eben. Beachte, dass der von `lock()` zurückgelieferte Shared-Pointer ein temporäres Objekt ist, das am Ende des vollständigen Ausdruckes wieder gelöscht wird.

5.5 Die Klasse `array`

Wie schon erwähnt, ist es nicht sinnvoll rohe Arrays zu verwenden. Besser ist es auf die Möglichkeiten der Standardbibliothek zurückzugreifen.

Prinzipiell kann die Klasse `vector` verwendet werden, die jedoch über einen gewissen Overhead verfügt, da diese Klasse über zusätzliche Features verfügt, wie zum Beispiel, dass die Größe verändert werden kann. Das bedeutet, dass diese Klasse Sinn macht, wenn man einen Container benötigt, dessen Größe sich ändern kann.

Will man jedoch lediglich eine einfache, sichere Möglichkeit, um auf Arrays eines bestimmten Typs zuzugreifen, dann bietet sich an, die Klasse `array` aus der Standardbibliothek zu verwenden.

```
1  #include <iostream>
2  #include <array>
3
4  using namespace std;
5
6  int main() {
7      array<int, 10> arr;
8      cout << "size: " << arr.size() << endl;
9  }
```

Wir sehen hier, dass es bei einem `array` notwendig ist, die Größe des Arrays als Template-Argument in die spitzen Klammern mit aufzunehmen. Das liegt daran, dass Templates vom Compiler direkt zur Übersetzungszeit umgesetzt werden und es damit de facto keinen Overhead im Vergleich zu einem rohen Array gibt!

Der Zugriff auf die einzelnen Elemente erfolgt genauso wie bei einem rohen Array oder einem Vektor:

```
1  for (auto i : arr) {
2      cout << i << " ";
3  }
```

Die Ausgabe kann folgendermaßen aussehen:

```
1  size: 10
2  -1079045240 134514728 1 65535 -1217703472 -1220706901
3  -1219111904 -1220706922 1 134514939
```

Hier sieht man, dass die Elemente einer Instanz von `array` ebenfalls nicht initialisiert werden. Der Grund sind wiederum Performanceüberlegungen. Initialisierung eines `array`s geht wieder mit der einheitlichen Initialisierung:

```
1  array<int, 10> arr2{1, 2, 3};
2  cout << "size: " << arr2.size() << endl;
3
4  for (auto i : arr2) {
5      cout << i << " ";
6  }
```

Die diesbezügliche Ausgabe sieht dann folgendermaßen aus:

```
1  size: 10
2  1 2 3 0 0 0 0 0 0 0
```

Wir sehen, dass jetzt die ersten drei Feldelemente mit den angegebenen Werten aus der einheitlichen Initialisierung initialisiert sind und der Rest des Arrays mit Nullwerten aufgefüllt wurde.

Der Zugriff über die Notation `[]` ist ebenfalls sehr effizient, überprüft aber genauso wenig auf die Feldgrenzen wie dies auch bei der Klasse `vector` nicht überprüft worden ist:

```
1  cout << endl;
2
3  cout << arr2[10000] << endl;
```

Diese Codezeile wird mit großer Wahrscheinlichkeit zu einem Absturz aufgrund eines Speicherzugriffsfehlers führen. Teste!

Ändere jetzt diese Zeile wie folgt ab:

```
1  cout << arr2.at(10000) << endl;
```

Auch hier wird es zu einem Absturz kommen, allerdings mit einer Exception vom Typ `std::out_of_range`, die leicht abgefangen werden kann:

```
1  try {
2      cout << arr2.at(10000) << endl;
3  } catch (const out_of_range& ex) {
4      cout << "error 'out of range': " << ex.what() << endl;
5  }
```

Exception werden wir uns im Abschnitt **9.2** noch detailliert ansehen.

Man könnte unter Umständen argumentieren, dass man sowieso die `[]` Notation verwenden könnte, da beide Fälle zu einem Abbruch geführt haben. Dies ist jedoch nicht so, da es zwei gravierende Unterschiede gibt:

- Der Abbruch über einen Adressbereichsfehler ist nicht abzufangen.
- Ein nicht korrekter Zugriff muss nicht unbedingt zu einem Adressbereichsfehler führen! Schauen wir uns diesen Fall an:

```
1  cout << arr[10] << endl;
```

Diese Anweisung gibt bei mir in diesem Fall einfach die Zahl 1 aus, obwohl der Zugriff nicht korrekt ist! Selbstverständlich kann es auch zu einer beliebigen anderen Ausgabe kommen oder auch einen Absturz nach sich ziehen. Eine Änderung zu `arr.at(10)` hätte dies erkannt!

Ein weiterer Vorteil an Instanzen von `array` ist, dass diese – im Gegensatz zu rohen Arrays – leicht kopiert und verglichen werden können:

```
1  if (arr == arr2)
2      cout << "arr == arr2" << endl;
3  else
4      cout << "arr != arr2" << endl;
5
6  arr = arr2;
7
8  if (arr == arr2)
9      cout << "arr == arr2" << endl;
10 else
11     cout << "arr != arr2" << endl;
```

Diese Anweisungen werden zu folgender erwarteter Ausgabe führen:

```
1  arr != arr2
2  arr == arr2
```

6 Funktionen

Die prinzipielle Deklaration und Definition einer Funktion in C++ haben wir schon kennengelernt. In diesem Kapitel geht es darum, Funktionen im Detail kennenzulernen.

In C++ werden Methoden einer Klasse ebenfalls als Funktionen, und zwar genauer als *member functions* (dt. so viel wie Mitgliedsfunktionen) bezeichnet. Wir werden diese entweder als Methoden oder als Member-Funktionen bezeichnen. Dieses Kapitel beschränkt sich auf „normale“ Funktionen, während Methoden im Abschnitt 8.2 auf der Seite 240 behandelt werden.

6.1 Deklaration und Definition

Den Aufbau einer Deklaration haben wir schon im Abschnitt 3.5 kennengelernt. Hier werden wir uns jetzt dem genauen Aufbau einer Funktionsdeklaration und einer Funktionsdefinition widmen.

Eine Funktionsdeklaration spezifiziert in jedem Fall zumindest den Typ des Rückgabewertes, den Namen der Funktion und die Reihenfolge und Typen der Parameter. Damit schauen einfache Beispiele folgendermaßen aus:

```
1  // functions.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  double squared(double);
7  void print(string msg);
8  int one();
9
10 int main() {
11 }
```

Die erste Deklaration legt als Typ eine Funktion `squared` an, die einen `double` als Parameter hat und einen `double` als Rückgabewert zurückliefert. Der Name der Parameter muss in einer Deklaration nicht zwingend angegeben werden.

Die Funktionsdeklaration `print` legt fest, dass die Funktion einen Parameter vom Typ `string` bekommt, der den Namen `msg` hat und keinen Wert zurückliefert, da `void` angegeben worden ist und `void` in diesem Fall bedeutet, dass es keinen Rückgabewert gibt.

Die letzte Deklaration legt eine Funktion `one` fest, die keine Parameter erhält und einen Wert vom Typ `int` zurückliefert. Beachte, dass hier *keine* Variable vom Typ `int` angelegt wird, die mit dem Nullwert initialisiert wird! Eine solche Variable hätte besser mit `int one{};` angelegt werden sollen. Damit sieht man wieder, dass man besser nur die einheitliche Initialisierung verwenden sollte.

So eine Funktionsdeklaration wird in C++ auch als Prototyp oder Funktionsprototyp (engl. *prototype*) bezeichnet.

Sonst tut dieses Programm natürlich nichts, da die Definition von `main` keinerlei Anweisungen enthält. Das Aufrufen der Funktionen wird allerdings nicht funktionieren, da die Funktionsdefinitionen noch fehlen. Das bedeutet, dass die nachfolgende Funktion `main` zwar vom Compiler korrekt übersetzt werden kann, aber der Linker die Adressen der Funktionen nicht kennt und deshalb eine Fehlermeldung generieren wird:

```
1  int main() {
2      print("2^2 - 1 = ");
3      cout << squared(2) - one() << endl;
4  }
```

Im Link-Vorgang, dem Verbinden der Bezeichner mit den eigentlichen Speicherobjekten, wird eine Fehlermeldung generiert werden, die besagt, dass es undefinierte Referenzen zu den Funktionen gibt. Probiere es aus!

Deshalb werden wir die folgenden Funktionsdefinitionen *nach* den Deklarationen aber *vor* dem Beginn der Funktion `main` einfügen:

```
1  double squared(double val) {
2      return val * val;
3  }
4
5  void print(string message) {
6      cout << message;
7  }
8
9  int one() {
10     return 1;
11 }
```

Hier sehen wir, dass Funktionsdefinitionen nicht mit einem Strichpunkt abgeschlossen werden. Weiters bemerken wir, dass der Name des Parameters der Funktion `print` sich von dem entsprechenden Namen in der Deklaration unterscheidet. Das ist kein Problem, da der Name der Parameter nicht zum Typ der Funktionsdeklaration gehört.

Jetzt können wir unser Programm wieder testen!

Allgemein gilt, dass es für eine Funktionsdeklaration genau eine Definition geben muss, die mit dem Typ der Deklaration übereinstimmt. Allerdings darf es mehrere Deklarationen zu einer Definition geben, die jedoch alle übereinstimmen müssen (siehe Abschnitt **3.4**).

In unserem obigen Beispiel hätten wir die Deklaration einfach weglassen können, da eine Definition ja eine spezielle Art der Deklaration ist. Da ein Bezeichner ab dem Zeitpunkt der Deklaration zur Verfügung steht, sind wir in der Lage die Funktionsdefinitionen von `squared`, `print` und `one` *hinter* `main` anzuordnen:

```
1  double squared(double);
2  void print(string msg);
3  int one();
4
5  int main() {
6      // wie vorher
7  }
8
9  double squared(double val) {
10     return val * val;
11 }
12
13 // restliche Funktionsdefinitionen folgen hier
```

Damit funktioniert unser Beispiel wie zuvor, da der Compiler innerhalb von `main` den Typ der Bezeichner `squared`, `print` und `one` kennt und den entsprechenden Code für den Funktionsaufruf generieren kann. Nach `main` findet der Compiler die Funktionsdefinitionen, vergleicht diese mit den Deklarationen und generiert den Funktionscode. Normalerweise verbindet der Linker der Compilersuite im Link-Vorgang auch gleich den Funktionsaufruf mit der Funktionsdefinition.

Natürlich wäre es in diesem Fall einfacher, die Funktionsdefinitionen vor der Funktion `main` anzuschreiben und keine extra Funktionsdeklarationen anzuschreiben. Allerdings benötigt man diese Funktionsdeklarationen sehr wohl in größeren Programmen. Wir werden dies im Abschnitt 7.1.2 besprechen.

Es besteht die Möglichkeit den Typ des Rückgabewertes einer Funktion durch den Compiler bestimmen zu lassen. Das macht vor allem bei der Verwendung von Templates einen Sinn, aber wir wollen uns hier die entsprechende Möglichkeit ansehen:

```
1  // sum.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  auto sum(int a, int b) -> decltype(a + b) {
7      return a + b;
8  }
9
10 int main() {
11     cout << "Die Summe von 3 und 5 ist " << sum(3,5) << endl;
12 }
```

Hier wird das Schlüsselwort `auto` für den Rückgabewert der Funktionsdefinition verwendet und weiters nach der Parameterliste mittels `->` der Typ angegeben. Zwar könnte man in diesem speziellen Fall auch direkt `-> int` schreiben, aber das macht keinen Sinn. `decltype()` ermittelt den Typ des übergebenen Ausdruckes und in unserem konkreten Fall ergibt sich der Rückgabewert als der Typ des Ausdruckes `a + b`. Da sowohl `a` als auch `b` den Typ `int` aufweisen, ist der Typ von `a + b` ebenfalls `int`.

C++14

In C++14 geht es sogar noch einfacher! Hier kann man nämlich den Teil `->` weglassen, wie das nachfolgende Beispiel zeigt.

```
1  #include <iostream>
2
3  using namespace std;
4
5  auto sum(int a, int b) {
6      return a + b;
7  }
8
9  int main() {
10     cout << "Die Summe von 3 und 5 ist " << sum(3,5) << endl;
11 }
```

Wichtig ist lediglich, dass bei mehrfachen Vorkommen einer `return` Anweisung alle Ausdrücke der `return` Anweisungen den gleichen Typ haben müssen. Auch rekursive Funktionen sind auf diese Art möglich, nur muss die `return`-Anweisung mit einem rekursiven Funktionsaufruf die letzte `return`-Anweisung innerhalb der Funktionsdefinition sein.

Hier muss allerdings aufgepasst werden, dass `auto` bei Forward-Deklaration durchaus eingesetzt werden kann, aber die Funktion kann erst dann verwendet werden, wenn diese *definiert* ist:

```
1  auto sum(int a, int b);
```

Damit haben wir den grundlegenden Aufbau einer Funktionsdeklaration und einer Funktionsdefinition kennengelernt.

Zu erwähnen ist, dass Funktionen innerhalb von anderen Funktionsdefinitionen deklariert, aber nicht definiert werden dürfen. Eine Funktionsdefinition innerhalb einer Funktionsdefinition ist also syntaktisch nicht möglich.

Weiters gibt es noch weitere Spezifikationssymbole (engl. *specifier*), die für eine Funktion verwendet werden können, die an den Anfang einer Funktionsdeklaration gestellt werden können.

Für „normale“ Funktionen, die nicht Methoden sind, gibt es die folgenden Spezifikationssymbole, die am Beginn der Funktion angegeben werden können:

- Mit `inline` wird dem Compiler mitgeteilt, dass er den Funktionsrumpf direkt einfügen soll. Dies ist im Abschnitt **7.2.2.3** beschrieben.
- `constexpr` gibt an, dass diese Funktion in einen einem Ausdruck verwendet werden kann, der selbst als `constexpr` gekennzeichnet ist. Dies ist im Abschnitt **7.2.2.4** beschrieben.
- Mittels `static` wird angegeben, dass eine Funktion interne Bindung hat. Das werden wir uns im Abschnitt **7.2.2.1** ansehen.
- `extern` gibt an, dass die Funktion externe Bindung aufweist. Dies ist im Abschnitt **7.2.1** beschrieben.

Weiters gibt es `noexcept`, das an das Ende der Funktion gesetzt werden kann und angibt, dass die Funktion keine Exception wirft. Dies wird im Abschnitt **9.3** beschrieben.

6.2 Funktionsaufruf

Der Aufruf einer Funktion ist je nach verwendetem System unterschiedlich. Meist werden die Argumente am Stack übergeben und der Rückgabewert entweder ebenfalls mittels dem Stack an den Aufrufer zurückgegeben oder in einem Register zurückgeliefert.

Bemerkung

Im Fall der oft verwendeten Intel-Prozessorfamilie und der C-Aufrufkonvention „cdecl“ sieht dies so aus, dass die Argumente zuerst auf den Stack platziert werden, dann wird die „CALL“ Anweisung aufgerufen, die zuerst die Rücksprungadresse ebenfalls auf den Stack gibt und danach zu der Adresse der Funktion verzweigt. Die Funktion kann auf die einzelnen Parameter zugreifen und speichert den Rückgabewert in ein Register. Danach wird die „RET“ Anweisung ausgeführt, die sich vom Stack die Rücksprungadresse holt und zu dieser zurück verzweigt. Der Aufrufer setzt den Stackpointer zurück und entfernt damit implizit die Argumente vom Stack.

6.2.1 Argumente

Die Parameter einer Funktion in der Funktionsdeklaration werden auch als formale Parameter (engl. *formal parameters*) bezeichnet. Beim Aufruf der Funktion werden die tatsächlichen Parameter (engl. *actual parameter*) der Funktion übergeben. Die tatsächlichen Parameter werden auch Argumente (engl. *arguments*) oder oft auch aktuelle Parameter genannt.

Es gibt prinzipiell zwei Arten der Übergabe von Parametern:

- mittels Kopien als Werte (engl. *per-value*)
- mittels Referenzen (engl. *per-reference*)

Wir werden auf die beiden Arten jetzt genauer eingehen.

6.2.1.1 Übergabe als Kopie

Schauen wir uns dazu nochmals unser Programm `functions.cpp` an und im speziellen die Übergabe der Parameter an die Funktionen. In dieser Datei haben wir die folgenden Funktionen mit Parameterübergabe als Kopie verwendet:

```
1  double squared(double val) {
2      return val * val;
3  }
4
5  void print(string message) {
6      cout << message;
7  }
```

In dieser Form – ohne weitere syntaktische Angabe – werden in C++ die Argumente und der Rückgabewert immer per-value übergeben. Dass die Argumente per-value übergeben werden, hat die Auswirkung, dass diese Argumente innerhalb der Funktion verändert werden können und keinen Nebeneffekt für den Aufrufer zur Folge haben.

Im konkreten Fall der Funktion `squared` bedeutet dies, dass innerhalb von `squared` der Parameter `val` geändert werden kann und dies keinerlei Auswirkung auf den Aufrufer hat. Ändere dazu deine Datei `functions.cpp` wie folgt ab:

```
1  double squared(double val) {
2      val = val * val;
3      return val;
4  }
5
6  int main() {
7      // ...
8      double val{5};
9      cout << squared(val) << endl;
10     cout << val << endl;
11 }
```

Die Ausgabe wird folgendermaßen aussehen:

```
1  25
2  5
```

Das bedeutet, dass mittels einer Übergabe als Kopie der Aufrufer von Nebeneffekten einer aufgerufenen Funktion geschützt ist. Dies ist logischerweise nur gültig, solange wir die Änderungen an dem Argument selbst betrachten und nicht über ein etwaig per Pointer referenziertes Speicherobjekt. Werden Pointer als Kopie übergeben, dann führen Änderungen am Pointer ebenfalls zu keinen Nebeneffekten. Änderungen an Speicherobjekten, auf die Pointer verweisen, werden vom Aufrufer der Funktion sehr wohl wahrgenommen:

```
1  void incr(double* pd) {
2      pd = nullptr;
3  }
4
5  int main() {
6      // ...
7      double counter{1};
8      double* pd{&counter};
9      incr(pd);
10     cout << ((pd == nullptr) ? 0 : *pd) << endl;
11 }
```

Die Ausgabe wird wie erwartet der aktuelle Wert 1 der Variable `counter` sein. Ändern wir jetzt die Funktion `incr` wie folgt ab:

```
1  void incr(double* pd) {
2      ++*pd;
3  }
```

Als Ausgabe erhalten wir jetzt natürlich 2.

Das Übergeben der Argumente per-value ist semantisch gesehen wie die Initialisierung einer Variable mit einem Wert zu betrachten. Wie wir später noch sehen werden bedeutet dies, dass bei Instanzen von Klassen der Kopierkonstruktor aufgerufen wird.

Wie schon erwähnt werden rohe Arrays als Parameter und rohe Arrays als Rückgabewert immer als Pointer auf das erste Arrayelement (per value) übergeben. Das hat Auswirkungen:

- Die Angabe der Feldgrenze bei der Parameterangabe ist unerheblich und hat außer zu Dokumentationszwecken keinerlei Auswirkung!

```
1  // array_param.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  void print(char* names[10]) {
7      cout << sizeof(names) << endl;
8  }
9
10 int main() {
11     char* names[10];
12     print(names);
13 }
```

Die Ausgabe dieses Programmes ist von der entsprechenden Architektur und im speziellen von der Größe eines Pointers abhängig! Ein System mit 4 Byte großen Zeigern liefert eben den Wert 4.

Achtung auch die Änderung der Variable `names` wie folgt hat keine Auswirkung auf die Übersetzung oder die Ausführung:

```
1  char* names[5];
```

Wir sehen, dass der Compiler hier keinerlei Überprüfungen vornimmt und aus dieser Sicht auch nicht vornehmen kann.

- Damit einhergehend ergibt sich, dass man über keinerlei Information verfügt, über wie viele Elemente das Array in der Funktion verfügt. Innerhalb von `main` weiß dies der Compiler aber schon! Schreibe die Funktionen `print` und `main` wie folgt um:

```
1  void print(const char* names[10]) {
2      cout << sizeof(names) << endl;
3  }
4
5  int main() {
6      const char* names[5] {"otto", "ulli", "susi"};
7      cout << sizeof(names) << endl;
8  }
```

Als genauer Beobachter ist dir wahrscheinlich aufgefallen, dass die Deklaration der Zeiger in der Parameterspezifikation von `print` als auch in der Definition von `names` jetzt mit einem `const` versehen worden ist.

Das liegt daran, dass `names` mit einer Liste von C-String-Literalen initialisiert worden ist. Diese sind vom Typ her ein Array von `const char` Elementen und haben, in diesem Beispiel, jeweils den Typ `const char[5]`. Damit es zu keiner Warnung seitens des Compilers kommt, wurde der Typ von `names` entsprechend umgeändert. Damit einhergehend wurde auch der Typ des Parameters der Funktion `print` geändert, auch wenn diese derzeit nicht mehr aufgerufen wird.

Unter der Voraussetzung, dass ein Pointer auf unserem System die Größe von 4 Bytes aufweist, wird es zu folgender Ausgabe kommen:

```
1  20
```

Das Array hat eine Größe von 5 Elementen mit je einem Pointer, woraus die Ausgabe von 20 folgt.

Dies kann man sich zunutze machen, um über die Elemente des Arrays zu iterieren. Hänge folgende Zeilen an:

```
1  for (int i{}; i < sizeof(names) / sizeof(const char*); ++i)
2      cout << names[i] << endl;
```

Durch die Division der Größe des Arrays durch die Größe eines einzelnen Elementes kommt man eben zur Anzahl der Elemente in dem Array.

Die Ausgabe, die die Schleife erzeugt, wird folgendermaßen aussehen:

```
otto
ulli
susi
```

Stimmt das? Wie viele Elemente haben wir in dem Array? Fünf Elemente, aber nur drei Ausgaben. Das liegt daran, dass es sich bei den letzten beiden Elementen um Null-Pointer handelt. Die Ausgabe eines Null-Pointers auf einen C-String macht natürlich keinen Sinn und versetzt den Ausgabekanal `cout` in einen Fehlerzustand. Das bewirkt, dass keine weiteren Ausgaben möglich sind, bevor der Ausgabekanal wieder zurückgesetzt wurde. Dies kann man leicht durch das Anfügen der folgenden Ausgabeoperation im Anschluss an die Schleife überprüfen:

```
cout << "Hello!" << endl;
```

Es wird zu keiner Ausgabe kommen! Es bleibt also die Möglichkeit, entweder die Ausgabe eines Null-Pointers zu unterbinden (z.B. mittels einer Abfrage) oder eben den Fehlerzustand zurückzusetzen. Füge deshalb die folgende Anweisung vor der Ausgabe von „Hello“ ein und die Ausgabe wird durchgeführt werden:

```
cout.clear();
```

Selbstverständlich wissen wir, dass dies durchaus einfacher gegangen wäre:

```
for (auto n : names) {
    if (n)
        cout << n << endl;
}
```

Auch hier darf man nicht vergessen, dass die beiden letzten Array-Elemente jeweils einen Null-Pointer darstellen und daher den Ausgabekanal in einen

Fehlerzustand versetzen würden. Deshalb wurde in diesem Beispiel überprüft, ob es sich um keinen Null-Pointer handelt.

Will man das Array an die Funktion `print` übergeben, dann muss man die Anzahl der Elemente als Parameter mitgeben. Ändere daher die Funktionsdefinition wie folgt ab:

```
1  void print(const char* names[], int size) {
2      cout << sizeof(names) << endl;
3      cout << size << endl;
4  }
```

Weiters muss noch ein entsprechender Aufruf hinzugefügt werden:

```
1  print(names, 3);
```

Hier haben wir nur die Anzahl der tatsächlich enthaltenen Elemente übergeben und damit wird auch die Ausgabe durch die Funktion `print` in unserem Fall erwartungsgemäß folgendermaßen aussehen:

```
1  4
2  3
```

Das Iterieren über die Elemente kann jetzt ganz leicht mit einer `for`-Zählschleife erreicht werden:

```
1  void print(const char* names[], int size) {
2      for (int i{}; i < size; ++i) {
3          cout << names[i] << endl;
4      }
5  }
```

Beachte, dass eine „foreach“ Schleife innerhalb der Funktion `print` aus den genannten Gründen nicht funktioniert. Der Compiler wird eine derartige Funktion `print` nicht einmal übersetzen:

```
1  void print(const char* names[], int size) {
2      for (auto n : names)
3          cout << n << endl;
4  }
```

Damit sehen wir, dass es sinnvoll ist, anstatt der rohen Arrays entweder die Klasse `array` oder die Klasse `vector` der Standardbibliothek zu verwenden.

6.2.1.2 Übergabe als Referenz

Die zweite Art der Übergabe von Parametern ist „per-reference“. Da wir zwei Arten von Referenzen in C++ kennengelernt haben, gibt es auch beide Arten bei der Parameterübergabe.

Mittels einer lvalue-Referenz kann man Daten einer Funktion übergeben und auch gleichzeitig aus dieser zurückliefern. Wir haben dies schon im Abschnitt 5.3.2 im Kontext der Besprechung der rvalue-Referenzen gesehen.

Hier folgt nochmals ein direktes Beispiel, das die direkte Übergabe von Daten und das Zurückliefern zeigt:

```
1  // lvaluerefpar.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  void incr(int& counter) {
7      ++counter;
8  }
9
10 int main() {
11     int counter{};
12
13     incr(counter);
14
15     cout << counter << endl;
16 }
```

Im Gegensatz zur Übergabe per-value wird hier nicht der Wert der Variable `counter` als Kopie übergeben, sondern faktisch die Adresse der Variable. Innerhalb der Funktion wird überall, ganz im Sinne der Verwendung der Referenz, eine implizite Dereferenzierung durchgeführt. Damit bewirkt der Inkrementoperator auch eine Erhöhung der Variable `counter` aus der Funktion `main`, womit es eben zur Ausgabe 1 kommt.

Die Übergabe per-reference sollte nur verwendet werden, wenn man Objekte in einer Funktion verändern und auch wieder zurückliefern möchte. Will man aus Performancegründen große Objekte ebenfalls als Referenz übergeben, dann sollte dies als konstante Referenz erfolgen.

Es ist generell anzuraten Daten im Normalfall per-value zu übergeben, da Nebeneffekte vermieden werden und damit eine klarere Struktur des Programmes entsteht.

Damit macht es in diesem konkreten Beispiel keinen Sinn die Variable `counter` per-reference zu übergeben.

Die zweite Art der Übergabe per-reference, ist die Übergabe als rvalue-Referenz. Betrachten wir nochmals unser Beispiel von vorhin und fügen der Funktion `main` die folgende Anweisung hinzu:

```
1  incr(2);
```

Damit wird der Compiler das Programm nicht mehr übersetzen können, da es sich bei dem Zahlenliteral 2 um einen rvalue handelt. Erinnere dich: Kann eine Adresse ermittelt werden, dann ist es ein lvalue, außer die Deklaration kennzeichnet ein `const` Speicherobjekt.

Um das Programm in dieser Form übersetzen zu können, füge die folgende Funktionsdefinition gleich hinter der Definition von `incr` an:

```
1  void incr(int&& counter) {
2      ++counter;
3  }
```

Mit dieser zusätzlichen Definition wird der Compiler jetzt zufrieden sein, da wir jetzt eine zusätzliche Definition einer Funktion `incr` haben, die eine rvalue-Referenz als formalen Parameter hat und sich beim Aufruf einen rvalue erwartet. Das

macht natürlich in unserem konkreten Fall absolut keinen Sinn, da die Erhöhung der lokalen Variable `counter` in der Funktion `incr` keinerlei Auswirkung hat! Damit sehen wir, dass die Verwendung von rvalue-Referenzen als formale Parameter eher selten Verwendung finden wird. Diese kommen hauptsächlich bei der Verwendung von benutzerdefinierten Datentypen zum Einsatz.

Wir sehen hier außerdem, dass wir Funktionen genauso wie Operatoren überladen können und werden uns das Überladen von Funktionen noch im Abschnitt **6.3** auf der Seite **187** ansehen.

Alternativ hätten wir den Compiler auch noch mit folgender Funktionsdefinition zufriedenstellen können:

```
1 void incr(const int& counter) {
2 }
```

Auch dann hätte das Übersetzen problemlos funktioniert, da die gleichen Regeln zutreffen wie schon im Abschnitt **5.3.1** besprochen. Es ist offensichtlich, dass auch dieser Ansatz keine Lösung ist, da wir innerhalb der Funktion `incr` den Wert der lokalen Variable `counter` nicht mehr verändern können, da dieser als `const` markiert ist.

6.2.1.3 Default-Argumente

Will man für einen Parameter keinen Wert mitgeben, dann kann man einen Default-Wert bei der Deklaration mitgeben.

```
1  // defaultargs.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int incr(int=0);
7
8  int incr(int counter) {
9      return counter + 1;
10 }
11
12 int main() {
13     cout << incr() << endl;
14     cout << incr(1) << endl;
15 }
```

Wir sehen bei diesem Beispiel, dass wir die Funktion sowohl mit als auch ohne Argument aufrufen können. Wird die Funktion ohne Argument aufgerufen, dann wird eben der Default-Wert verwendet. Damit kommt es zu folgender Ausgabe:

```
1  1
2  2
```

Wichtig ist, dass Default-Werte nur für die letzten Parameter eingesetzt werden können. Das bedeutet, dass nach einem Default-Parameter kein nicht Default-Parameter verwendet werden kann.

Ein Default-Argument kann im gleichen Scope weder wiederholt noch geändert werden. Damit sind solche Deklarationen fehlerhaft:

```
1  int incr(int=0);
2  int incr(int=0);  // darf nicht wiederholt werden
3  int incr(int=1);  // darf nicht geändert werden
```

Die Default-Argumente werden zum Zeitpunkt des Funktionsaufrufes ausgewertet:

```
1  int getDefault();
2
3  int incr(int=getDefault());
```

Das heißt, dass beim Aufruf von `incr` in diesem Fall zuerst die Funktion `getDefault` aufgerufen werden wird.

6.2.1.4 Variable Anzahl an Parametern

Es gibt in C++ mehrere Möglichkeiten wie eine variable Anzahl an Argumenten übergeben werden kann:

- Es gibt die Möglichkeit die Parameterliste mittels `...` abzuschließen. Das bedeutet, dass eine variable Anzahl an beliebigen Parametern an die Funktion übergeben werden kann. Diesen Ansatz werden wir nicht weiter betrachten, da dieser nicht typsicher ist.
- Weiters gibt es die Möglichkeit mittels Funktionstemplates eine typsichere Übergabe von einer variablen Anzahl an Parametern zu erreichen. Dies werden wir uns im Abschnitt **13.2** ansehen.
- Man kann Parameter eines bestimmten Typs mittels des schon bekannten Typs `initializer_list` erreichen. Diesen Ansatz werden wir uns jetzt ansehen:

```
1  // vararginitlist.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  void log(initializer_list<string> messages) {
7      for (auto msg : messages) {
8          cout << msg << endl;
9      }
10 }
11
12 int main() {
13     log({"testing", "warning", "error"});
14 }
```

Wir sehen, dass wir auch hier eine variable Anzahl von Argumenten des gleichen Typs haben.

6.2.2 Rückgabewert und Beendigung einer Funktion

Der Rückgabewert einer Funktion ist analog zu der Parameterübergabe zu betrachten.

Das Zurückliefern von Referenzen ist in bestimmten Situationen sinnvoll und wird oft in Zusammenhang mit benutzerdefinierten Datentypen eingesetzt. Ein bekanntes Beispiel kennen wir schon, nämlich bei der Verwendung des Operators `<<` im Zusammenhang mit der Ausgabe und dem Objekt `cout`. Der Operator liefert in diesem Fall eine Referenz auf die eigene Instanz, also `cout`, zurück. Damit können Methodenaufrufe verkettet werden:

```
1  cout << "abc" << "def" << endl;
```

Dies funktioniert so, dass `cout << "abc"` eine Referenz auf `cout` zurückliefert und hiermit für den Aufruf des Operators `<<` mit dem Operanden `"def"` zur Verfügung steht. Mit `endl` funktioniert dies wieder analog auch wenn dieser Rückgabewert in dem Beispiel keine weitere Verwendung findet.

Es gibt mehrere Möglichkeiten wie eine Funktion beendet werden kann:

- Es wird eine `return`-Anweisung ausgeführt so wie wir es schon öfters gesehen haben.
- Das Ende der Funktion mit der geschwungenen Klammer wird erreicht und der Typ des Rückgabewertes ist mit `void` angegeben.
- Es wird eine Exception (Ausnahme) geworfen. Das werden wir uns im Abschnitt **9.2** ansehen.
- Innerhalb des Funktionsrumpfes wird ein Systemaufruf (engl. *system call*, eingedeutscht Systemcall) für eine das Programm beendende Aktion aufgerufen, wie zum Beispiel `exit()`. Mehr dazu siehe Abschnitt **7.3**.

  Für derartige Funktionen, die nicht mehr aus der Funktion zurückspringen, wie zum Beispiel `exit()`, kann man ein *Attribut* `[[noreturn]]` voranstellen, das genau dies angibt. Der Vorteil liegt darin, dass einerseits dieses Verhalten hiermit dokumentiert ist und andererseits der Compiler einen effizienteren

Code generieren kann. Dies sieht dann im Fall der Funktion `exit` folgendermaßen aus:

```
1  [[noreturn]] void exit(int);
```

Sollte die Funktion trotzdem zurückspringen, dann ist das Verhalten nicht definiert.

Attribute kann man syntaktisch gesehen weitgehend überall im Programmtext anschreiben, allerdings sind nur ganz wenige Attribute im Standard festgeschrieben.

C++14

In C++14 wurde ein weiteres Attribut eingeführt, das es erlaubt Definitionen als veraltet zu markieren. Damit wird der Compiler bei der Verwendung eine Warnung erzeugen, wie das nachfolgende Beispiel zeigt:

```
1  // deprecated.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  [[deprecated]]
7  int x{1};
8
9  [[deprecated]]
10 int f(int x) {
11     return x;
12 }
13
14 int main() {
15     cout << f(x) << endl;
16 }
```

6.2.3 Lokale Variable

Lokale Variablen werden in Funktionen genauso behandelt wie „normale“ lokale Variablen. Interessant in diesem Zusammenhang ist, dass lokale Variablen als „statisch“ deklariert werden können:

```
1  // staticvar.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int nextid() {
7      static int lastid;
8
9      return ++lastid;
10 }
11
12 int main() {
13     for (int i{}; i < 3; ++i)
14         cout << nextid() << endl;
15 }
```

Die Ausgabe wird folgendermaßen ausfallen:

```
1  1
2  2
3  3
```

Eine mit `static` deklarierte lokale Variable behält ihren Wert über mehrere Funktionsaufrufe hinweg. Außerdem wird eine derartige Variable, im Gegensatz zu lokalen ohne `static`, immer initialisiert. Wird kein Wert als Initialisierung angegeben, dann wird der „Null-Wert“ genommen.

6.3 Überladen von Funktionen

Prinzipiell haben wir schon gesehen, dass wir Funktionen überladen können. Das bedeutet, dass wir mehrere Funktionsdeklarationen mit dem gleichen Namen,

aber unterschiedlichen Parametern haben können. Der Compiler wird in Abhängigkeit der Anzahl und der Typen der Argumente die richtige Funktion beim Aufruf auswählen:

```
1  // functionoverloading.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  void sayHello(char c) {
7      cout << "Hello " << c << "!" << endl;
8  }
9
10 void sayHello(const char* str) {
11     cout << "Hello " << str << "!" << endl;
12 }
13
14 void sayHello(string str) {
15     cout << "Hello " << str << "!" << endl;
16 }
17
18 int main() {
19     sayHello('!');
20     sayHello("World");
21     string name{"Bob"};
22     sayHello(name);
23 }
```

Die Ausgabe wird natürlich folgendermaßen aussehen:

```
1  Hello !!
2  Hello World!
3  Hello Bob!
```

Es ist sehr schön zu sehen, dass C++ in Abhängigkeit des Typs des Arguments, jeweils die richtige Funktion zur Ausführung bringt.

Die Regeln, die der Reihe nach angewandt werden, um die richtige Funktion auszuwählen, sind in etwa folgendermaßen:

a. Gibt es eine exakte Übereinstimmung der Anzahl, Reihenfolge und Typen der Argumente mit denen der Parameter der Funktionsdeklaration, dann wird die entsprechende Funktion verwendet. Für das gerade gezeigte Beispiel trifft diese Regel zu.

b. Gibt es eine Übereinstimmung nach der Durchführung von Promotions, dann wird die entsprechende Funktion verwendet. Promotions wurden schon im Abschnitt **3.10** erklärt.

c. Danach wird versucht mit einer Konvertierung (siehe Abschnitt **3.10**) eine Funktion zu finden.

d. Im Anschluss wird noch versucht mit einer benutzerdefinierten Konvertierung zu einer Übereinstimmung zu kommen.

e. Führten alle vorangegangen Versuche nicht zu einem Ergebnis, dann werden noch Funktionsdeklarationen mit einer variablen Anzahl von Parametern auf Basis von `...` herangezogen.

Bei all diesen Regeln ist zu beachten, dass die Reihenfolge der Funktionsdeklarationen keine Rolle spielt. Weiters spielt der Typ des Rückgabewertes der Funktion keine Rolle.

Außerdem werden Funktionen aus verschiedenen Scopes *nicht* für das Überladen in Betracht gezogen! Wir haben schon im Abschnitt **3.6** gesehen, dass Bezeichner des aktuellen Geltungsbereiches die Bezeichner der umschließenden Geltungsbereiche überschatten. Das kann durchaus zu überraschenden Ergebnissen führen:

```
1  // overloadingscope.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int incr(int counter) {
7      cout << "int" << endl;
8      return counter + 1;
9  }
10
11 int incr(double counter) {
12     cout << "double" << endl;
13     return counter + 1;
14 }
15
16 int main() {
17     int incr(double counter);
18
19     incr(1);
20 }
```

Die Ausgabe wird in diesem Fall der String `double` sein, obwohl es sich bei 1 um ein Zahlenliteral vom Typ `int` handelt! Durch die Deklaration der Funktion `int incr(double)` steht nur diese Funktion in diesem Scope zur Verfügung. Es wird mittels einer Promotion `int` zu einem `double` implizit konvertiert und daher kann diese Funktion beim Aufruf verwendet werden.

Hier noch einmal: Funktionen, die in verschiedenen Geltungsbereichen deklariert sind, überladen sich nicht! Besonders wichtig ist das im Zusammenhang mit Vererbung wie im Abschnitt **12.2** gezeigt wird.

Eine einfache Abhilfe in diesem konkreten Beispiel ist, dass die Funktion aus dem globalen Scope aufgerufen wird. Das kann ganz leicht erreicht werden indem der Bereichsauflösungsoperator `::` eingesetzt wird. Ersetze dazu `incr(1)` durch `::incr(1)` und als Ausgabe wird daher `int` erscheinen.

Auch wenn dies wahrscheinlich nicht oft vorkommen wird, hat dies jedoch eine wichtige Anwendung im Zusammenhang mit benutzerdefinierten Datentypen. Wir werden uns dies noch im Abschnitt **12.2** ansehen.

6.4 Übergeben von Funktionen

Manchmal macht es Sinn, eine Funktion an eine Funktion zu übergeben. Die übergebene Funktion kann innerhalb der Funktion aufgerufen werden. Damit kann man nicht nur einzelne Daten übergeben, sondern auch Verhalten mitübergeben.

Es gibt die folgenden Möglichkeiten:

- Pointer auf eine Funktion (siehe Abschnitt 6.4.1)
- Funktionsobjekte (siehe Abschnitt 6.4.2)
- Lambdafunktionen (siehe Abschnitt 6.4.3)

6.4.1 Pointer auf Funktion

Die einfachste Möglichkeit ist, Funktionen direkt zu übergeben. In diesem Sinne wird die Übergabe einer Funktion genauso behandelt, wie die Übergabe eines rohen Arrays.

Schauen wir uns vorerst folgendes kleines Programm an:

```
1  #include <iostream>
2
3  using namespace std;
4
5  double add(double a, double b) {
6      return a + b;
7  }
8
9  double mul(double a, double b) {
10     return a * b;
11 }
12
13 int main() {
14     double (*f)(double, double);
15
16     f = add;
17     cout << f(3, 2) << endl;
18     f = mul;
19     cout << f(3, 2) << endl;
20 }
```

Die Ausgabe sieht dann so aus:

```
5
6
```

- Zuerst werden zwei, vom Typ gleiche, Funktionen `add` und `mul` definiert, die die übergebenen Parameter addieren beziehungsweise multiplizieren und das Ergebnis jeweils zurückliefern.
- Innerhalb von `main` wird zuerst eine Variable `f` definiert, die als Typ einen Pointer auf eine Funktion darstellt. Diese Funktion, auf die `f` zeigt, bekommt zwei Parameter von Typ `double` und liefert einen Wert vom Typ `double` zurück.

  Wichtig sind die Klammern um `*f`, da der Operator `()` eine höhere Priorität hat als der Operator `*`!
- Die Zuweisung von `add` auf `f` zeigt sehr schön, dass die Funktion `add` als Pointer auf eine Funktion betrachtet wird. In C++ werden solche Funktionen als Pointer betrachtet, die einen Typ haben, der dem Prototyp der Funktion entspricht.
- Beim Aufrufen einer Funktion sehen wir, dass der Pointer direkt beim Aufruf als normaler Funktionsaufruf verwendet werden kann.

Da Funktionen in diesem Sinne als Pointer aufgefasst werden können, können diese Pointer auch an andere Funktionen übergeben werden.

```
using func = double (*)(double, double);

double accumulate(initializer_list<double> list, func f) {
    double res{};
    for (auto elem : list) {
        res = f(res, elem);
    }
    return res;
}
```

Und innerhalb der Funktion `main` füge folgende Anweisung hinzu:

```
1  cout << accumulate({1.0, 2.0, 3.0, 4.0}, add) << endl;
```

Die Ausgabe ergibt in diesem Fall 10. Wir sehen, dass wir das Verhalten der Funktion `accumulate` verändern können, indem wir verschiedene Funktionen übergeben.

6.4.2 Funktionsobjekt

Alternativ zu den Funktionszeigern besteht die Möglichkeit, Funktionsobjekte zu verwenden. Ein Funktionsobjekt ist an sich ein normales Objekt, das jedoch über einen Operator `operator()` verfügt. Das bedeutet, dass man Objekte mit einem solchen Operator wie eine normale Funktion aufrufen kann.

Mittels der Standardbibliothek kann man leicht Funktionsobjekte anlegen:

```
1  // funcobj.cpp
2  #include <iostream>
3  #include <functional>
4
5  using namespace std;
6
7  double add(double a, double b) {
8      return a + b;
9  }
10
11 double mul(double a, double b) {
12     return a * b;
13 }
14
15 int main() {
16     function<double(double, double)> f;
17     f = add;
18     cout << f(3, 2) << endl;
19 }
```

Soweit funktioniert das Beispiel genauso wie vorher. Wo ist jetzt der Vorteil? Prinzipiell kann mit diesen Objekten jede beliebige Funktionsart verwendet werden. In dem obigen Beispiel wurde `f` ein Pointer auf eine Funktion zugewiesen. Aber es ist auch möglich `f` eine Lambda-Funktion (siehe Abschnitt **6.4.3**) oder eine Methode zuzuweisen.

Weiters kann man auch einzelne oder alle Parameter mit Werten vorbelegen. Hänge folgende Zeilen an dein Programm an:

```
1  auto f2 = bind(f, 3, 2);
2  cout << f2() << endl;
```

Auch hier ist die Ausgabe natürlich 5. Willst du nicht alle Parameter mit Werten binden, sondern nur einzelne, dann funktioniert das folgendermaßen:

```
1  using namespace std::placeholders;  // _1, _2,...
2  auto f3 = bind(f, _1, 2);
3  cout << f3(3) << endl;
```

Du siehst, dass auf diese Weise der erste Parameter der „neuen“ Funktion `f3` an den ersten Parameter der Funktion `f` gebunden wird, während der zweite Parameter von `f` mit dem fixen Wert 2 belegt wird.

Hier folgt jetzt noch ein Beispiel, das zeigt wie man die Reihenfolge der Parameter beim Belegen ändern kann. Füge dazu folgende Funktion zu deinem Programm hinzu, das das Volumen eines Quaders bestehend aus den Seiten a, b und c berechnet:

```
1  double volume_cuboid(double a, double b, double c) {
2      cout << "a = " << a << ", b = " << b << ", c = "
3           << c << endl;
4      return a * b * c;
5  }
```

Im `main` füge die beiden folgenden Zeilen hinzu:

```
auto f4 = bind(volume_cuboid, _2, 2, _1);
f4(3, 4);
```

Das wird zu folgender Ausgabe führen, die klar zeigt, wie sich der Einfluss der Parameter _1 und _2 auf den Aufruf auswirkt:

```
a = 4, b = 2, c = 3
```

Weiters kann man auch einen eigenen benutzerdefinierten Datentyp definieren, den man verwenden kann, um Funktionsobjekte zu verwenden. Dies werden wir uns im Abschnitt **11.4.1** ansehen.

6.4.3 Lambda-Funktionen

Will man eine Funktion schreiben, die eigentlich nur an einer Stelle verwendet wird, dann kann man eine Lambdafunktion verwenden. Es handelt sich dabei um eine anonyme Funktion, um eine Funktion ohne Namen.

```
// lambdaexpr.cpp
#include <iostream>
#include <algorithm>

using namespace std;

int main() {
    auto values = {1, 2, 3, 4};
    int sum{};
    for_each(begin(values),
             end(values),
             [&sum](int val) { sum += val; });
    cout << sum << endl;
}
```

Dieses Programm wird als Ausgabe 10 liefern. Es funktioniert folgendermaßen:

- `#include <algorithm>` inkludiert den Teil der Standardbibliothek, der die Algorithmen enthält. Algorithmen sind ein wichtiger Teil der Standardbibliothek und werden später noch detailliert besprochen werden.

  In diesem konkreten Fall geht es darum die Funktion `for_each` zugreifbar zu machen, die einen Iterator auf den Beginn von `values`, einen Iterator auf das Ende von `values` und eine Lambdafunktion als Parameter bekommt. Iteratoren sind ein weiterer wichtiger Bestandteil der Standardbibliothek und werden ebenfalls später noch im Detail besprochen.

  Diese Funktion `for_each` funktioniert so, dass sie für jedes Element von `values` beginnend mit `begin(values)` bis exklusive `end(values)`, also für alle Elemente von `values`, die Lambdafunktion aufruft.

- In der Zeile 12 wird eine Lambdafunktion verwendet, die per Referenz auf die lokale Variable `sum` zugreifen kann und weiters bei jedem „Schleifendurchgang“ das aktuelle Element als Parameter `val` bekommt. Im Rumpf der Lambdafunktion wird das aktuelle Element zur Variable `sum` aufsummiert.

Eigentlich handelt es sich bei der Definition einer Lambdafunktion um einen Ausdruck. Das ist auch deshalb leicht zu erkennen, da die Lambdafunktion als Parameter an die Funktion `for_each` übergeben wird.

Ein Lambdaausdruck wird von dem Compiler in ein Funktionsobjekt übersetzt und besteht syntaktisch aus den folgenden Teilen:

a. Der erste Teil wird als „capture“ (bedeutet so viel wie „erfassen“) bezeichnet und gibt an, auf welche Variable der Rumpf der Lambdafunktion zugreifen kann. Dieser Teil ist an den eckigen Klammern zu erkennen und muss angegeben werden.

   Es gibt hier die folgenden Möglichkeiten:

   – Es wird eine leere Capture-Liste verwendet. Das bedeutet, dass die eckigen Klammern leer sind, also `[]`. Damit sind keine lokalen Variablen aus dem umschließenden Kontext im Rumpf der Lambdafunktion zugreifbar. Die einzige Möglichkeit Daten im Rumpf der Lambdafunktion zu verwenden ist auf die Parameter zuzugreifen.

   – Die Capture-Liste enthält nur das Zeichen `=`, also sieht dieser Teil so aus: `[=]`. Das bedeutet, dass alle lokalen Variablen verwendet werden können und diese als Kopie (per-value) zur Verfügung stehen.

– In der Capture-Liste ist nur das Zeichen `&` enthalten. Damit wird ausgedrückt, dass alle lokalen Variablen als Referenz zur Verfügung stehen.

  Werden lokale Variablen als Referenz im Rumpf der Lambdafunktion verwendet, dann ist zu beachten, dass diese Lambdafunktion nur verwendet wird, wenn die lokalen Variablen noch existieren!

  Das folgende Beispiel ist in diesem Sinne fehlerhaft:

```
1  // lambdacapture.cpp
2  #include <iostream>
3  #include <vector>
4  #include <functional>
5
6  using namespace std;
7
8  int main() {
9      vector<function<string()>> v;
10     {
11         string name{"Bob"};
12         v.push_back([&name]() { return name; });
13     }
14     cout << v[0]() << endl;
15 }
```

  Beim Aufruf mittels `v[0]()` steht die lokale Variable `name` nicht mehr zur Verfügung! Das Verhalten ist undefiniert!

– Die Capture-Liste enthält einzelne Namen von lokalen Variablen. Diese stehen als Kopie zur Verfügung, wenn diesen nicht ein `&` vorangestellt wird.

  Beispiel: `[a, &b, c]`

– Die Capture-Liste beginnt mit einem `=` und es folgen weitere Namen von lokalen Variablen. Das bedeutet, dass alle lokalen Variablen als Kopie zur Verfügung stehen, außer die in der Liste angeführten Namen, die als Referenz vorhanden sind.

  Beispiel: `[=, a, b, c]`

– Als alternative zum vorhergehenden Punkt kann die Capture-Liste mit einem `&` beginnen. Damit stehen alle lokalen Variablen als Referenz zur

Verfügung außer denjenigen Namen, die in der folgenden Liste angeführt sind und als Kopie verwendet werden können.

Beispiel: `[&, a, b, c]`

b. Danach folgt die Liste der Parameter wie man es von einer Funktion gewohnt ist.

c. Anschließend kann ein optionales Spezifikationssymbol `mutable` folgen, das angibt, dass die lokalen Variablen, die als Kopie zur Verfügung stehen verändert werden können.

Die folgenden Anweisungen werden vom Compiler nicht übersetzt werden:

```
1 string name{"Bob"};
2 auto f = [name]() { name = ""; };
```

Ändert man den Lambdaausdruck ab, sodass dieser `mutable` enthält, kann der Compiler dies übersetzen:

```
1 auto f = [name]() mutable { name = ""; };
```

Die Idee dieses Ansatzes ist, dass man normalerweise den Zustand (also die Instanzvariablen) eines Funktionsobjektes nicht verändert, sondern diesen gleich wie beim Anlegen des Objektes belässt. Aus diesem Grund ist dies die Default-Einstellung von solchen generierten Funktionsobjekten indem der Compiler einen Operator `operator()` generiert, der als `const` gekennzeichnet ist und daher den Zustand nicht ändern kann. Wir werden uns dies im Abschnitt ?? ansehen.

d. Optional kann ein `noexcept` folgen. Die Bedeutung dieses Spezifikationssymbols werden wir im Abschnitt **9.3** kennenlernen.

e. Optional kann der Rückgabetyp mittels `->` erfolgen.

f. Dann folgt der Rumpf der Lambdafunktion in geschwungenen Klammern.

C++14

C++14 geht einen Schritt weiter und erlaubt auch generische Lambdafunktionen! Das sind solche Lambdafunktionen, die auch generische Parameter erlauben. Darunter werden Parameter verstanden, die mittels `auto` deklariert werden. Siehe folgendes Beispiel.

```
1  // lambdageneric.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      auto f = [](auto x) { return x; };
8      cout << f(1) << endl;
9      cout << f("abc") << endl;
10     cout << f(false) << endl;
11 }
```

Sinnvoll einsetzen lässt sich diese Erweiterung in Verbindung mit Templates (siehe Abschnitt **13.2.1**).

C++14

Weiters wurden in C++14 die Behandlung der Capture-Liste erweitert. In C++11 werden die Parameter der Liste entweder als Kopie oder als (lvalue-)Referenz übergeben. Damit ist es nicht möglich einen rvalue in die Capture-Liste aufzunehmen. Diese Einschränkung wurde in C++14 aufgehoben, wie das folgende Beispiel zeigt.

```
1  // lambdacaptureexpr.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      auto f = [x = 1]() { return x; };
8      cout << f() << endl;
9  }
```

Interessant wird dies allerdings erst im Zusammenhang mit `move` oder bei der Verwendung von Funktionen in der Capture-Liste:

```
1  // lambdacapturemove.cpp
2  #include <iostream>
3  #include <memory>
4
5  using namespace std;
6
7  int main() {
8      std::unique_ptr<int> pi(new int(1));
9      auto f = [x = std::move(pi)]() { return *x; };
10     cout << f() << endl;
11     auto g = []() { return 0; };
12     auto h = [x = g()] { return x; };
13     cout << h() << endl;
14 }
```

7 Modularisierung

Ein Programm stellt eine Lösung für ein Problem dar. Bei wachsender Problemgröße ergibt sich jedoch, dass der Umfang und die Komplexität der Problemlösung ebenfalls größer wird.

Daraus ergeben sich die folgenden Konsequenzen:

- Meist kann ein großes Problem nicht in einem Lösungsschritt gelöst werden. Stattdessen wird das Problem sinnvollerweise in Teilprobleme zerlegt, die einzeln gelöst werden. Die Lösung des Gesamtproblems ergibt sich durch Zusammensetzen der Teillösungen.

- Selbst wenn eine Lösung in einem Schritt für ein großes Problem möglich wäre, wäre diese Lösung in der Regel ebenfalls groß und damit unübersichtlich. Mit der fehlenden Übersichtlichkeit ergeben sich beim Programmieren mehr Fehlerquellen und die Wartbarkeit des Programmes nimmt ab.

- Mit zunehmender Problemgröße kann ein Problem innerhalb eines vorgegebenen Zeitrahmens von einem einzelnen Entwickler nicht gelöst werden. Damit werden mehrere Entwickler benötigt, die sinnvollerweise jeder an einer eigenen Teillösung arbeiten.

- Hat man ein Teilproblem einmal gelöst, dann kann man diese Lösung unter Umständen auch in einem anderen Kontext wiederverwenden.

Aus diesen Gründen heraus, versucht man ein Programm in kleinere Teile aufzuteilen. Jeder kleine Programmteil löst für sich ein Teilproblem des gesamten Problems. Solche kleineren Teile werden Module genannt (siehe Abschnitt **7.1**).

Mit zunehmender Anzahl an Modulen wächst auch die Anzahl der Bezeichner, die in einem Programm zur Verfügung stehen. Bei einer großen Anzahl an Bezeichnern, liegt wiederum eine Unübersichtlichkeit vor und die Gefahr von Namenskollisionen steigt. Deshalb gibt es das Konzept des Namensraumes, mit dem man eine Strukturierung der Identifier vornehmen kann (siehe Abschnitt **7.4**).

Ein Programm besteht aus einzelnen Modulen, die miteinander verbunden werden und einen gewissen Aufbau aufweisen.

7.1 Module

Ein Modul stellt seine Funktionalität über eine Schnittstelle zur Verfügung, während die Implementierung der Funktionalität dem Benutzer verborgen bleibt. Oft verwendet ein Modul andere Module, um die geforderte Funktionalität erbringen zu können.

Derzeit bietet C++ kein eigenes Konzept an, das Module direkt unterstützt. Stattdessen wird ein Modul auf der Basis von *physischer Strukturierung* des Quelltextes und sprachlichen Hilfsmitteln gebildet.

- In den meisten C++ Implementierungen stehen uns Dateien zur *physischen Strukturierung* des Quelltextes zur Verfügung. An sich geht der C++ Standard nicht explizit von Dateien als Container für den C++ Quelltext aus, da der C++ Standard offen lässt wie C++ Quelltext repräsentiert und gespeichert wird. Da die meisten C++ Implementierungen Dateien verwenden, werden wir weiterhin von Dateien sprechen.

  Das heißt, dass ein Modul in C++ seine Funktionalität in einer `.cpp`-Datei ablegt und seine Schnittstelle durch eine Headerdatei (siehe Abschnitt **7.1.2**) zur Verfügung stellt. Headerdateien haben in der Regel eine Erweiterung `.h`, allerdings sind prinzipiell beliebige Erweiterungen möglich.

- Als *sprachliche Hilfsmittel* stehen prinzipiell Variablen, Funktionen, Klassen und Namensräume zur Verfügung, die in einer Headerdatei angeführt werden und die Schnittstelle beschreiben.

  Als Hilfsmittel für die generische Programmierung stehen Templates zur Verfügung (siehe Abschnitt **13.1**).

  Weiters gibt es Spezifizierungssymbole wie `inline`, `extern`, `static`, `const` und `constexpr`, die für eine genauere Schnittstellenbeschreibung herangezogen werden.

Nimmt ein Modul andere Module in Anspruch, dann inkludiert es deren Schnittstellen wiederum als Headerdateien. Solch eine Headerdatei enthält hauptsächlich Funktionen, Klassen und Objekte.

Die Implementierung des Moduls kann man vor dem Benutzer des Moduls verbergen, sodass dieser nur die Schnittstelle zu Gesicht bekommt.

7.1.1 Übersetzung zu einem Programm

Der Übersetzungsschritt einer einzelnen Datei sieht so aus, dass der Compiler zuerst in einem Vorverarbeitungsschritt den Präprozessor (von engl. *preprocessor*, Vorverarbeiter) beauftragt, die Quelldatei in eine sogenannte Übersetzungseinheit (engl. *translation unit*) zu übersetzen.

Bei dieser Vorverarbeitung werden zuerst alle Präprozessoranweisungen ausgeführt und als Ergebnis entsteht reiner C++ Quelltext ohne Präprozessoranweisungen. Wir erkennen solche Präprozessoranweisungen an dem Rautezeichen `#`. Mehr dazu im Abschnitt **7.1.2**.

Danach wird die entstandene Übersetzungseinheit vom eigentlichen Compiler in eine Objektdatei übersetzt. Diese enthält nur Objektcode (auch Maschinencode genannt), also Anweisungen in der Maschinensprache des Prozessors.

Im Zuge dieses Vorganges müssen die Teilergebnisse, wie zum Beispiel die Objektdateien, nicht unbedingt als eigenständige Dateien vorliegen.

Sind alle benötigten Schritte vorgenommen worden, dann geht es weiter mit dem sogenannten Linken (engl. *link*, binden). In diesem Schritt, der durch einen Linker (engl. *linker* oder *link editor*) vorgenommen wird, werden alle Bezeichner von Deklarationen mit den entsprechenden Definitionen – unter Umständen in anderen Objektdateien vorhanden – verbunden. Schauen wir uns die folgende einfache Anweisung zur Ausgabe eines Textes an:

```
1  cout << "Test" << endl;
```

Hier werden die Bezeichner `cout` und `endl` verwendet, die in der Headerdatei `iostream` deklariert worden sind. Allerdings verweisen diese Bezeichner auf Objekte, für die es auch eine eindeutige Definition geben muss, die in diesem konkreten Fall in der Standardbibliothek enthalten ist. Diese Standardbibliothek wird von der C++ Implementierung zur Verfügung gestellt und auch automatisch verwendet.

Objektdateien können zu Bibliotheksdateien (engl. *library*) zusammengefasst werden und hiermit gemeinsam verwendet werden.

Prinzipiell können die zu verbindenden Bezeichner in beliebigen Objektdateien oder in beliebigen Bibliotheken enthalten sein. Wichtig ist nur, dass der Linker diese findet und die Adressen der Speicherobjekte den Bezeichnern zuordnen kann. Das heißt, der Linker fügt den gesamten benötigten Code in eine Datei

zusammen und ersetzt die Stellen der Bezeichner mit den Adressen der Speicherobjekte.

Damit haben wir – vereinfacht ausgedrückt – ein ausführbares Programm erhalten. Üblicherweise ist der Linker in den Compiler integriert!

7.1.2 Headerdateien

Werfen wir einen genaueren Blick auf den Präprozessor. Bis jetzt haben wir uns die `#include` Anweisung angesehen, die eine von mehreren Präprozessoranweisungen ist. Abgesehen von der „if“ und der „define“ Anweisung werden heute jedoch meist keine Präprozessoranweisung mehr verwendet.

Verwendet haben wir bis jetzt solche Präprozessoranweisungen nur um Headerdateien aus der Standardbibliothek einzubinden, wie zum Beispiel:

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

Diese Anweisung bindet den gesamten Text der Headerdatei `iostream` anstelle dieser Präprozessoranweisung ein. Unter Umständen enthält eine Headerdatei wiederum Präprozessoranweisungen, dann werden diese im Zuge der Einbindung vom Präprozessor entsprechend ausgeführt.

Wie schon erwähnt haben die Headerdateien meist die Dateinamenerweiterung `.h`, dies muss jedoch nicht zwingend so sein. Das beste Beispiel sind die Headerdateien der C++ Standardbibliothek, die überhaupt keine Erweiterung aufweisen, wie im obigen Beispiel zu sehen ist.

Es gibt zwei Arten wie die `#include`-Präprozessoranweisung verwendet werden kann:

- Einerseits kann diese in der schon bekannten Form `#include <iostream>` – also mit dem Headernamen in spitzen Klammern – verwendet werden. Dies bedeutet, dass der Präprozessor die angegebene Datei in den Standardverzeichnissen der C++ Implementierung suchen soll.

  Innerhalb der Standardbibliothek gibt es eine spezielle Notation für Headerdateien, die von der Programmiersprache `C` kommen und in C++ direkt verwendet werden können. Den Namen dieser Headerdateien ist ein kleines „c“ vorangestellt, wie wir dies schon von `<cstdint>` kennen. Auch die Bezeichner dieser Headerdateien sind im Namensraum `std` enthalten und müssen

deshalb zum Beispiel mittels `std::` oder `using namespace std;` zugreifbar gemacht werden.

- Andererseits kann man diese in der Form `#include "mathutils.h"` verwenden, also mit doppelten Anführungszeichen anstatt mit spitzen Klammern. Dies bedeutet, dass der Präprozessor die Datei in den Verzeichnissen des aktuellen Projektes suchen soll. Wird nichts weiter angegeben, dann bedeutet dies, dass im aktuellen Verzeichnis gesucht wird.

Nehmen wir einmal an, dass wir eine nützliche Funktion `squared` geschrieben haben, die das Quadrat der übergebenen Zahl berechnet und das Ergebnis zurückliefert und wir diese Funktion in mehreren Übersetzungseinheiten verwenden wollen. Dann macht es Sinn, diese Funktion in eine eigene Datei zu geben:

```
1 // mathutils.cpp
2 double squared(double val) {
3     return val * val;
4 }
```

Wir können diese Datei übersetzen. Allerdings müssen wir dazu dem Compiler mitteilen, dass dieser nur eine Objektdatei erzeugen soll, da dieser sonst ein ausführbares Programm generieren will. Dies ist jedoch nicht möglich, da die Funktion `main` nicht vorhanden ist und im Zuge des Bindens die Funktionsdefinition für `main` nicht zur Verfügung steht.

Verwende deshalb die entsprechenden Optionen deines Compilers oder konfiguriere deine Entwicklungsumgebung entsprechend. Für die Übersetzung eines Quelltextes in eine Objektdatei ist im Abschnitt **A.2** die Anleitung für die gebräuchlichen Compiler und Betriebssysteme vorhanden. Wir gehen jetzt davon aus, dass eine Objektdatei vorliegt.

In einem weiteren Schritt legen wir eine neue Datei an, die diese Funktion verwenden soll:

```
1  // main.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  int main() {
7      cout << squared(4) << endl;
8  }
```

Will man jetzt dieses Programm übersetzen, dann muss man dem Compiler (mit Linker) auch alle Quelldateien bekannt geben. Allerdings wirst du feststellen, dass eine Übersetzung nicht möglich ist, da der Bezeichner `squared` dem Compiler nicht bekannt ist. Das ist auch verständlich, da wir keine Deklaration und im Speziellen auch keine Definition dafür haben.

Die einfachste Möglichkeit dies zu lösen ist, dass die Definition dem Compiler bekannt gegeben wird. Modifiziere den Quellcode bezüglich der Präprozessoranweisungen folgendermaßen:

```
1  #include <iostream>
2
3  #include "mathutils.cpp"
```

Beim Übersetzen braucht nur die Datei `main.cpp` angegeben werden. Das funktioniert, aber es ist nicht gut, denn es wurde die *Definition* der Funktion zur Gänze in die Datei `main.cpp` eingebunden. Das führt in größeren Programmen unweigerlich zu Fehlern und dupliziertem Maschinencode. Eigentlich wollten wir, die Funktionsdeklaration der Funktion `squared()` dem Compiler beim Übersetzen der Datei `main.cpp` bekanntgeben. Lege deshalb eine Headerdatei `mathutils.h` mit folgendem Inhalt an, die die Schnittstelle unseres Moduls definiert:

```
1  double squared(double val);
```

Ändere weiters in der Datei `main.cpp` die Präprozessoranweisung für das Einbinden unserer Headerdatei wie nachfolgend ab.

```
1  #include "mathutils.h"
```

Du siehst hier, dass der Name der Headerdatei in doppelte Hochkommas eingeschlossen ist und nicht in spitzen Klammern wie bei den Headerdateien der Standardbibliothek.

Übersetze nun wiederum nur die Datei `main.cpp`. Das Übersetzen wird jetzt problemlos funktionieren, aber das Binden wird mit einer Fehlermeldung scheitern. Diese Fehlermeldung sagt aus, dass es eine undefinierte Referenz zu der Funktion `squared(double)` gibt.

Das ist leicht zu beheben, indem du auch die Datei `mathutils.cpp` dem Compiler mitgibst. Damit wird immer die Datei `main.cpp` als auch die Datei `mathutils.cpp` übersetzt. Es macht natürlich wenig Sinn alle Dateien immer zu übersetzen, wenn nur eine Datei verändert worden ist. Deshalb können beim Übersetzen anstatt der nicht geänderten Quelldateien die Objektdateien angegeben werden, der Compiler (mit integriertem Linker) wird nur die abgeänderten Quelldateien übersetzen und die Objektdateien binden.

Eine weitere Verbesserung werden wir jetzt einbauen, die uns in Zukunft vor (Tipp-)Fehlern schützen kann. Füge die gleiche Präprozessoranweisung `#include "mathutils.h"` auch an den Anfang der Datei `mathutils.cpp` ein. Damit kann der Compiler beim Übersetzen der Datei `mathutils.cpp` die Definitionen gegen die Deklarationen in der Datei `mathutils.h` prüfen. Damit ist sichergestellt, dass diese Deklarationen immer zu den Definitionen passen und beim Einbinden der Includedatei in andere Dateien wissen wir, dass die Deklarationen richtig sind.

Eigentlich ist es so, dass sogenannte Make-Systeme verwendet werden oder die Entwicklungsumgebung nach erfolgter Konfiguration sich selber um das Übersetzen und Linken der relevanten Dateien kümmert. Weiters wird es in der Regel so sein, dass mehrere zusammenhängende Objektdateien zu einer Bibliothek zusammengefasst werden, sodass nur mehr die Bibliothek beim Linken angegeben werden muss.

Das ist soweit schon einmal ganz gut, allerdings gibt es noch immer ein Problem, dass sich dadurch äußert, dass eine Headerdatei durchaus auch mehrmals eingelesen werden kann. Damit dies nicht passiert greift man zu folgender Technik, die als Wächter (engl. *guard*) bezeichnet wird. Dazu wird die Headerdatei so umgebaut, dass der gesamte Inhalt mit einer if-Präprozessoranweisung umschlossen wird:

```
1  #ifndef MATHUTILS_H
2  #define MATHUTILS_H
3
4  double squared(double val);
5
6  #endif
```

Das bedeutet, dass der Präprozessor den if-Zweig der `#ifndef` Anweisung (if not defined) nur betreten wird, wenn das Präprozessormakro `MATHUTILS_H` noch nicht definiert ist. In diesem Fall wird diese im if-Zweig definiert und die eigentlichen Deklarationen folgen. Abgeschlossen wird mit einem `#endif`. Wird diese Headerdatei ein zweites Mal beim Übersetzen eingelesen, dann ist das Präprozessormakro `MATHUTILS_H` schon definiert und die Deklarationen werden nicht beachtet.

Damit verhindern wir auch rekursives Einlesen von Headerdateien! Nehmen wir an, dass wir eine Headerdatei `a.h` und eine Headerdatei `b.h` haben. Dann könnte es sein, dass `a.h` die Datei `b.h` inkludieren will und `b.h` wiederum `a.h` inkludieren will. Ohne einen Wächter ergibt sich eine Endlosrekursion.

Im Abschnitt 7.5 wird festgehalten welcher Inhalt in Headerdateien enthalten sein darf und welcher nicht.

7.2 Binden

Wir haben schon gesehen, dass mehrere Übersetzungseinheiten durch den Linker gebunden werden. Jetzt werden wir uns ansehen, welche Möglichkeiten es gibt, Bezeichner miteinander zu binden.

Es gibt zwei Arten wie in C++ Bezeichner an Objekte gebunden werden können:

- Bei der externen Bindung (engl. *external linkage*) kann ein Bezeichner einer Übersetzungseinheit auch in anderen Übersetzungseinheiten verwendet werden.
- Bei der internen Bindung (engl. *internal linkage*) steht der Bezeichner nur innerhalb einer Übersetzungseinheit zur Verfügung.

7.2.1 Externe Bindung

Wenn wir uns diese Einteilung ansehen, dann erkennen wir, dass Funktionsdeklarationen offensichtlich in die Kategorie „external linkage“ fallen, sonst hätte das Binden des Funktionsaufrufes `squared(4)` zur Definition in der Objektdatei von `mathutils.cpp` nicht funktioniert.

Der Spezifizierer `extern` hat im Zusammenhang mit Funktionen die Bedeutung, dass die Funktion eine externe Bindung aufweist. Dies ist redundant, da eine Funktion ja automatisch eine externe Bindung aufweist.

Ebenfalls in diese Kategorie fallen die globalen Variablen! Erweitern wir dazu unser Beispiel um die Variable `pi`. In der Datei `mathutils.cpp` fügen wir eine globale Variable hinzu:

```
1 #include <cmath>
2
3 double pi{atan(1)*4};
```

In der zugehörigen Headerdatei fügen wir eine entsprechende Deklaration hinzu:

```
1 extern double pi;
```

Dann können wir in der Datei `main.cpp` auf `pi` zugreifen:

```
1 cout << pi << endl;
```

Das `extern` bedeutet, dass es sich hierbei um eine Deklaration handelt, die aussagt, dass sich die zugehörige Definition in einer anderen Übersetzungseinheit befindet. Ohne `extern` würde es sich um eine Definition handeln und es sind keine doppelten Definitionen von Bezeichnern erlaubt, die auf Speicherobjekte verweisen. Dies würde der Linker als Fehler melden.

Etwas komplizierter wird es bei Definitionen von Klassen, Templates und inline-Funktionen und dergleichen, denn hier gilt die sogenannte „one-definition rule“.

Prinzipiell ist es so, dass es exakt eine Definition einer Klasse, eines Templates,... innerhalb eines Programmes geben darf. Eine solche Definition darf nicht in mehreren Objektdateien enthalten sein. C++ kann dies in dieser Form allerdings nicht überprüfen.

Aus diesem Grund gibt es die „one-definition rule“ (ODR), die besagt, dass zwei Definitionen einer Klasse, eines Templates oder einer inline-Funktion als Exemplare einer eindeutigen Definition angesehen werden, wenn sich diese in verschiedenen Übersetzungseinheiten befinden und aus der Sicht der Programmiersprache C++ als gleich angesehen werden.

Es ist leider so, dass der Compiler diese Regel nicht absolut überprüfen kann (bei mehreren Übersetzungseinheiten) und es daher sinnvoll ist, seine Headerdateien geschickt zu verwenden sowie Module einzusetzen.

Hier folgt noch ein Beispiel, das diese Situation demonstriert. Nehmen wir an, wir schreiben ein Modul, das einen Record beinhalten soll, der aus einer `id` und `name` besteht. Weiters gibt es eine Funktion, die für einen Zeiger auf einen Record die `id` zurückliefert:

```
1  // record.h
2  struct Record {
3      T id;
4      char* name;
5  };
6
7  T get_id(Record* ptr);
```

Wir sehen hier, dass der Typ `T` bis jetzt nicht festgelegt wurde. Das wird jetzt ausgenutzt, um den „Fehler“ zu demonstrieren. Implementieren wir jetzt das Modul:

```
#include <iostream>

using T = std::string;

#include "record.h"

T get_id(Record* ptr) {
    return ptr->id;
}
```

Wir sehen hier, dass `T` als ein `std::string` festgelegt wurde und die Funktion `get_id` gemäß unserer Anforderung implementiert wurde.

Jetzt wollen wir dieses Modul verwenden:

```
// recorduser.cpp
#include <iostream>

using namespace std;

using T = char;
#include "record.h"

int main() {
    Record rec;
    rec.id = 'a';
    cout << get_id(&rec) << endl;
}
```

Das Programm wird wahrscheinlich fehlerfrei übersetzen, obwohl es die ODR verletzt! Allerdings wird das Programm beim Ausführen mit hoher Wahrscheinlichkeit mit einem Adressbereichsfehler abstürzen. Teste!

Dies liegt daran, dass wir den Typ `T` hier verschieden deklariert haben und die C++-Implementierung diesen Fehler nicht erkennen konnte. Natürlich ist es in diesem Fall einfach, den Fehler zu vermeiden, da man lediglich den Typ von `T` in der Headerdatei `record.h` deklarieren hätte müssen.

7.2.2 Interne Bindung

7.2.2.1 `static`-Variable und `static`-Funktion

Globale Variablen, die mittels `static` markiert sind, weisen eine interne Bindung auf. `static` ist in diesem Zusammenhang ein Relikt aus der Programmiersprache `C` und wird bei globalen Variablen dazu verwendet, um die externe Bindung in eine interne Bindung zu wandeln.

```
1 static int num_people;
```

Damit ist die globale Variable `num_people` nur mehr in der aktuellen Übersetzungseinheit sichtbar und wird vom Linker nicht gesehen. In Headerdateien haben solche Definitionen nichts zu suchen!

Funktionen, die mittels `static` gekennzeichnet werden, bekommen automatisch interne Bindung. Nehmen wir einfach an, dass wir unsere Funktion `squared` mit einem `static` versehen:

```
1 static double squared(double);
```

Die Wirkung ist, dass diese nur in der aktuellen Übersetzungseinheit zur Verfügung steht. Damit einhergehend macht es wieder keinen Sinn, so eine Funktionsdeklaration in eine Headerdatei zu geben.

7.2.2.2 Konstante

Wenn wir uns jetzt das Beispiel mit der Variable `pi` noch einmal ansehen, dann werden wir sicher feststellen, dass es sich bei `pi` weniger um eine Variable als mehr um eine Konstante handelt.

In `C++` kann eine Konstante zum Beispiel mit dem Schlüsselwort `const` definiert werden. Gehen wir also her und fügen an den Anfang der Definition von `pi` in der Datei `mathutils.cpp` das Schlüsselwort `const` hinzu:

```
1 const double pi{atan(1)*4};
```

Wenn du jetzt das Programm wieder übersetzen willst, dann wirst du mit einem Linker-Fehler konfrontiert werden. Das liegt daran, dass eine `const` Variable in C++ defaultmäßig interne Bindung aufweist. Es gibt prinzipiell zwei Möglichkeiten, wie man dies lösen kann:

- Man kann der Konstante bei der Definition explizit mittels `extern` eine externe Bindung zuweisen. Das bedeutet, dass vor `const` noch ein `extern` hinzugefügt werden muss:

```
1  extern const double pi{atan(1)*4};
```

  Durch `extern` wird die defaultmäßige interne Bindung von `const` auf externe Bindung umgewandelt.

- Alternativ dazu kann man die Konstante zur Gänze in die Headerdatei verschieben (ohne Angabe von `extern`). Daher gibt es jetzt nicht nur eine Konstante, sondern so viele Exemplare der Konstante wie es Übersetzungseinheiten gibt, die diese Konstante verwenden, also die Headerdatei `mathutils.h` einbinden.

  Das hat aber den Nachteil, dass es einerseits die Konstante mehrmals im Objektcode und damit auch im Hauptspeicher gibt. Weiters können sich auch leichter Fehler einschleichen, wenn man Änderungen in der Headerdatei vornimmt, aber „vergisst“ eine oder mehrere Quelldateien zu übersetzen, die diese Headerdatei einbinden. Damit kann es unterschiedliche Werte für die Konstante im laufenden Programm geben.

  Bei einer Verwendung eines korrekt konfigurierten Buildsystems wird dies nicht vorkommen und daher wird es auch nicht zu unterschiedlichen Werten kommen. Trotzdem wird es mehrere Speicherobjekte geben und daher werden wir diesen Ansatz auch nicht weiter betrachten.

Als Alternative zu `const` kann eine Konstante auch mit dem Schlüsselwort `constexpr` definiert werden. Bezüglich Bindung verhält sich `constexpr` wie `const` und kann auch so verwendet werden. Zwei spezielle Punkte sind jedoch zu beachten:

- Im Standard C++11 und auch in C++14 ist die Funktion `atan` nicht als `constexpr` markiert (zu `constexpr`-Funktionen siehe Abschnitt **7.2.2.4**) und deshalb kann diese Funktion eigentlich nicht zur Berechnung von π verwendet werden. Es ist allerdings so, dass bestimmte Compiler wie zum Beispiel

der GNU g++ in einer aktuellen Version die Funktion atan sehr wohl als constexpr deklariert und damit einer Verwendung in der Initialisierung der constexpr Konstante pi nichts im Wege steht.

- Da eine constexpr Konstante nicht notwendigerweise eine Speicheradresse besitzt, macht es hier eindeutig mehr Sinn, diese in einer Headerdatei zu platzieren!

7.2.2.3 inline-Funktionen

Entwickeln wir das Beispiel wieder ein bisschen weiter. Warum soll für die Berechnung des Quadrats, die nur aus einer einzelnen Multiplikation besteht, auch zusätzlich ein Funktionsaufruf durchgeführt werden, der einen gewissen Overhead erzeugt? Besser wäre es, wenn wir den Compiler dazu bewegen könnten, anstatt des Funktionsaufrufes die Berechnung an Ort und Stelle durchzuführen.

Dies kann man dadurch erreichen, dass man der Funktionsdefinition ein inline voranstellt, das dem Compiler nahe legt, dass der Inhalt der Funktion anstatt des Funktionsaufrufes eingefügt wird. Damit dies funktioniert, muss der Compiler bei jedem Übersetzen die Definition der Funktion zur Hand haben. Das bedingt, dass die Definition einer inline-Funktion in der Headerdatei vorhanden sein muss.

Verschiebe deshalb die Definition von squared in die Headerdatei, stelle ein inline voran und lösche die Deklaration. Damit ist in der Headerdatei mathutils.h folgendes bezüglich der Funktion squared enthalten:

```
1 inline double squared(double val) {
2     return val * val;
3 }
```

inline Funktionen weisen eine interne Bindung auf, obwohl es sich um eine Funktionsdefinition handelt. Analog dazu verhält es sich mit constexpr-Funktionen.

Der Sinn einer derartigen inline-Spezifikation liegt darin, dass der Aufruf einer Funktion zugunsten einer größeren Codemenge eingespart wird. Damit ergibt sich, dass so eine Art von Optimierung nur dann Sinn macht, wenn die Funktion relativ klein ist.

7.2.2.4 `constexpr`-Funktionen

Wie wir uns schon im Abschnitt 3.9 über `constexpr` Ausdrücke angesehen haben, kann eine Funktion in einem `constexpr` Ausdruck normalerweise nicht verwendet werden. Um eine Funktion in einem solchen Ausdruck verwenden zu können, muss diese ebenfalls mit `constexpr` markiert sein.

Schauen wir uns dazu die folgende Funktion `digitsum` an, die die Quersumme einer dezimalen Zahl berechnet:

```
1  // constexprfunc.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  constexpr unsigned int digitsum(unsigned int number) {
7      return (number > 9) ?
8          (number % 10) + digitsum(number / 10) : number;
9  }
10
11 int main() {
12     constexpr unsigned int number{digitsum(12345)};
13     cout << number << endl;
14 }
```

Die Ausgabe wird in diesem Fall `15` sein, da $1 + 2 + 3 + 4 + 5 = 15$ ist.

In diesem Fall wird die Funktion `digitsum` zur Übersetzungszeit durch den Compiler ausgewertet und nicht zur Laufzeit! Zur Laufzeit wird in diesem konkreten Fall lediglich eine Konstante ausgegeben, aber keine Berechnung durchgeführt.

Der Compiler überprüft, ob innerhalb eines `constexpr`-Ausdruckes nur `constexpr`-Konstanten und `constexpr`-Funktionen vorkommen. Du kannst dies leicht überprüfen, indem du `constepxr` bei der Definition der Funktion `digitsum` entfernst. Der Compiler wird eine Fehlermeldung liefern.

`constexpr`-Funktionen können auch außerhalb von `constexpr`-Ausdrücken verwendet werden. Ändere dazu die Funktion `main` wie folgt ab:

```
1  int main() {
2      unsigned int number{};
3      cin >> number;
4      cout << digitsum(number) << endl;
5  }
```

Damit wird die Funktion `digitsum` jetzt zur Laufzeit ausgeführt und nicht mehr zur Übersetzungszeit.

Allerdings kann eine `constexpr` Funktion in C++11 nur sehr einfach aufgebaut sein! Sie muss aus einer einzelnen `return`-Anweisung bestehen und darf auch keine Schleifen und keine lokalen Variablen beinhalten. Weiters darf so eine Funktion keine Nebeneffekte bewirken. Das bedeutet, dass eine `constexpr`-Funktion auf keine Variablen außerhalb schreibend zugreifen darf. Auch Veränderungen von Referenz-Parametern sind nicht erlaubt.

So eine Funktion wird an sich als „pure function“ bezeichnet. Also keine Änderungen und für die gleichen Argumente wird immer das gleiche Ergebnis zurückgeliefert.

C++14

In C++14 wurden diese strengen Auflagen gelockert. Es sind jetzt auch Schleifen und lokale Variablen erlaubt, wie du im nächsten Beispiel sehen kannst, das die kleinste Zahl aus allen übergebenen Zahlen bestimmt.

```cpp
#include <iostream>
#include <limits>

using namespace std;

constexpr int min(std::initializer_list<int> numbers) {
    int min = std::numeric_limits<int>::max();
    for (int n : numbers)
        if (n < min) min = n;
            return min;
}

int main() {
    constexpr int min_number = min({5, 1, 8, -3, 9});
    cout << min_number << endl;
}
```

Analog zu `constexpr`-Konstanten und `inline`-Funktionen, weisen auch `constexpr`-Funktionen interne Bindung auf.

7.3 Programm

Ein Programm ist eine Sammlung kompilierter Übersetzungseinheiten, die der Linker zu einer ausführbaren Datei bindet. Jedes Programm muss genau eine Funktion `main` enthalten, die eine der folgenden Formen aufweist:

- `int main() { /* ... */ }`
- `int main(int argc, char* argv[]) { /* ... */}`

Der Rückgabewert wird an das aufrufende Programm weitergegeben. Jeder Wert ungleich Null wird in der Regel als Fehler interpretiert.

Ein Programm wird durch eine der folgenden Möglichkeiten beendet:

- Wenn die Funktion durch die Funktion `exit(int)` aus der Headerdatei `<cstdlib>` der Standardbibliothek aufgerufen wird. Damit wird das Programm an dieser Stelle beendet.

Dies wird als normale Beendigung interpretiert und es wird eine ganze Zahl als Argument mitgegeben, die als Rückgabewert an das aufrufende Programm zurückgegeben wird.

Bei Aufruf der Funktion `exit()` werden einige Aufräumarbeiten durchgeführt. Es werden die Destruktoren der `static`-Variablen und der thread-lokalen Variablen aufgerufen, jedoch nicht die der lokalen Variablen. Außerdem werden geöffnete Dateien geschlossen.

Es besteht außerdem die Möglichkeit, dass man Funktionen mittels `atexit()` registrieren kann, die durch `exit()` zur Ausführung gebracht werden. Damit kann man abschließende Aktionen erzwingen.

Schauen wir uns vorerst einmal das folgende Programm an:

```
1  #include <iostream>
2  #include <cstdlib>
3
4  using namespace std;
5
6  struct Variable {
7      string name;
8      ~Variable() {
9          cout << name << " var destructed" << endl;
10     }
11 };
12
13 int main() {
14     Variable local_var{"local"};
15     static Variable static_var{"static"};
16 }
```

Damit wird es zu folgender erwarteter Ausgabe kommen:

```
1  local var destructed
2  static var destructed
```

Rufen wir jetzt die Funktion `exit()` innerhalb von `main` auf:

```
1  exit(EXIT_SUCCESS);
```

Damit wird der Prozess mit einem implementierungsabhängigen Rückgabewert beendet, der auf der jeweiligen Plattform als Erfolg interpretiert wird.

Dann werden wir feststellen, dass für die lokale Variable `local_var` nicht mehr der Destruktor aufgerufen wird!

Als Nächstes können wir noch einen „exit handler“ in Form einer Funktion `exit_handler` schreiben:

```
1  void exit_handler() {
2      cout << "exit handler..." << endl;
3  }
```

Diese Funktion muss noch registriert werden, sodass diese beim Beenden des Programmes aufgerufen wird. Füge deshalb den folgenden Code vor den Aufruf der Funktion `exit` ein:

```
1  int result = atexit(exit_handler);
2
3  if (result != 0) {
4      cout << "exit_handler registration failed" << endl;
5      return EXIT_FAILURE;
6  }
```

Damit wird die Funktion `exit_handler` als „exit-handler“ registriert und beim Beenden des Programmes aufgerufen. Analog zu `EXIT_SUCCESS` gibt es auch ein `EXIT_FAILURE`, das in diesem konkreten Fall aufgerufen wird, wenn die Registrierung fehlschlägt.

Will man, dass überhaupt keine Destruktoren aufgerufen werden, dann kann anstatt `exit` die Funktion `quick_exit(int)` verwendet werden. Analog zu `atexit()` steht in diesem Fall `at_quick_exit()` zur Verfügung.

- Wenn die Funktion `main` beendet mittels der `return`-Anweisung und Angabe eines Rückgabewertes oder durch Erreichen des Ende der Funktion erreicht

wird. Wird das Ende der Funktion erreicht, dann wird der Rückgabewert 0 zurückgegeben.

Diese Art der Beendigung ist äquivalent zu der normalen Beendigung einer Funktion und anschließendem Aufruf von `exit` mit dem angegebenen Rückgabewert.

- Wenn die Funktion `abort()` aufgerufen wird. Dies gilt als Abbruch und es werden keinerlei Destruktoren aufgerufen und keine mittels `atexit()` oder `at_quick_exit()` registrierten Funktionen aufgerufen.

  Ob geöffnete Dateien geschlossen werden, ist von der konkreten Implementierung abhängig.

  Das aufrufende Programm erhält einen implementierungsabhängigen Rückgabewert, der eine fehlerhafte Beendigung anzeigen soll.

- Wenn eine Exception aufgetreten ist, die nicht abgefangen wurde. Anders ausgedrückt heißt dies, dass kein Exception-Handler diese Exception behandelt hat. Was dies genau bedeutet werden wir uns noch im Abschnitt **9.2** ansehen.

  Weiters wird das Programm beendet, wenn eine Exception innerhalb einer Funktion geworfen wird, obwohl die Funktion mit `noexcept` deklariert wurde (siehe Abschnitt **9.3**).

7.4 Namensraum

Die zugrunde liegende Problematik ist, dass mit der Anzahl der Bezeichner die Wahrscheinlichkeit steigt, dass sich Bezeichner überschneiden und man bei vielen Dateien nicht leicht herausfinden kann, wo ein Bezeichner deklariert ist.

In C++ wird mit einem Namensraum (engl. *namespace*) ein Geltungsbereich erzeugt, von dem von außerhalb mittels expliziter Qualifizierung auf die Bezeichner des Namensraumes zugegriffen werden kann.

7.4.1 Bereichsauflösungsoperator

Schauen wir uns einmal das folgende Beispiel an, das sich nicht übersetzen lässt:

```
1  // scoperesolution.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  double sum(double op1, double op2) {
7      return op1 + op2;
8  }
9
10 int main() {
11     double sum;
12     sum = sum(1, 2);
13     cout << sum << endl;
14 }
```

Der Fehler ist, dass die Definition der lokalen Variable `sum` die Definition der globalen Funktion überschattet (siehe Abschnitt **3.6**). Daher muss man dem Compiler explizit mitteilen, dass man die globale Funktion `sum` meint und nicht die lokale Variable mit dem gleichen Namen.

Das geht mit dem Bereichsauflösungsoperator (engl. *scope resolution operator*) `::` in der folgenden Art und Weise:

```
1  sum = ::sum(1, 2);
```

Dieser Bereichsauflösungsoperator gibt in dieser Form an, dass auf den globalen Namensraum zugegriffen werden soll.

7.4.2 Definition eines Namensraums

Stellen wir uns jetzt einmal vor, dass wir mehrere verschiedene Bibliotheken verwenden und jede Bibliothek uns eigene Headerdateien zur Verfügung stellt, die von uns alle inkludiert werden. Damit ist die Wahrscheinlichkeit relativ hoch, dass es zu Namenskollisionen im globalen Namensraum kommt!

Damit dies nicht passiert, könnte jede Bibliothek einen eigenen Namensraum definieren, in dem alle dessen Bezeichner verpackt werden. Ein Namensraum gruppiert also logisch zusammengehörige Bezeichner und es sind die Bezeichner der einzelnen Bibliotheken voneinander unabhängig.

Mit der Definition des folgenden Namensraums ist die Funktion `sum` nicht mehr im globalen Namensraum vorhanden. Ein Namensraum stellt deshalb einen eigenen Scope dar. Die folgende Definition legt einen benannten Namensraum an, der vorerst nur eine Funktion `sum` enthält:

```
1  // namespace.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  namespace MathUtils {
7      double sum(double op1, double op2) {
8          return op1 + op2;
9      }
10 }
```

Nur die folgende `main`-Funktion zu verwenden funktioniert jetzt nicht mehr, da es jetzt keine globale Funktion `sum` mehr gibt:

```
1  int main() {
2      double sum;
3      sum = ::sum(1, 2);
4      cout << sum << endl;
5  }
```

Auch `sum = sum(1, 2)` kann mit der gleichen Begründung natürlich nicht verwendet werden.

Um auf die Funktion jetzt zuzugreifen, muss auf eine der im Abschnitt **3.12** angeführten Möglichkeiten zurückgegriffen werden.

Zusätzlich besteht die Möglichkeit, direkt auf die Funktion `sum` des Namensraumes `MathUtils`, mittels des Bereichsauflösungsoperators, zuzugreifen:

```
1  cout << MathUtils::sum(1, 2) << endl;
```

Namensräume sind in C++ offen. Im folgenden Beispiel sieht man dies sehr schön, da mit den beiden `namespace`-Deklarationen sowohl die Funktion `sum` als auch die Funktion `mul` im Namensraum `MathUtils` enthalten sind:

```
1  namespace MathUtils {
2      double sum(double, double);
3  }
4
5  namespace MathUtils {
6      double mul(double, double);
7  }
```

Natürlich können diese Namensraumteile auf verschiedene Dateien verteilt werden.

Außerdem können Namensräume verschachtelt werden. Dies macht Sinn, wenn ein Namensraum sehr groß wird und sich in logisch zusammenhängende Teilnamensräume teilen lässt.

```
1  // namespace_nested.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  namespace MathUtils {
7      namespace Arithmetic {
8          double sum(double op1, double op2) {
9              return op1 + op2;
10         }
11     }
12 }
13
14 int main() {
15     cout << MathUtils::Arithmetic::sum(1, 2) << endl;
16 }
```

Es besteht natürlich auch eine gewisse Gefahr, dass verschiedene Bibliotheken die gleichen Bezeichner für ihre Namensräume verwenden. Daher ist es sinnvoll, diese Bezeichner relativ lange und aussagekräftig zu wählen. Damit einhergehend ist die Verwendung mittels des Bereichsauflösungsoperators natürlich mühsam.

Nehmen wir einmal an, dass ein Namensraum `Com_Musterfirma_MathUtils` heißt. Da hier offensichtlich der umgekehrte Domänenname als Anfang des Bezeichners dieses Namensraumes gewählt wurde, ist es relativ unwahrscheinlich, dass eine andere Bibliothek den selben Bezeichner verwendet hat. Der Nachteil ist, dass dieser Bezeichner lange und unhandlich ist.

Es besteht in C++ die Möglichkeit einen Namensraumalias zu definieren:

```
1  namespace MU = Com_Musterfirma_MathUtils;
2
3  cout << MU::Arithmetic::sum(1, 2) << endl;
```

7.4.3 Argument-Dependent Lookup

Nehmen wir an, dass eine Funktion `f` einen Parameter vom Typ `T` erwartet. Dann ist es meist so, dass diese Funktion im selben Namensraum definiert ist wie der

Typ `T`. Aufgrund dieser Überlegung, wird in C++ eine Funktion `f`, wenn diese nicht im aktuellen Scope oder den überliegenden Scopes gefunden wird, ebenfalls im Scope des Typs `T` gesucht. Diese Regel wird als „Argument-Dependent Lookup“ (ADL, dt. argumentabhängiges nachschlagen) bezeichnet.

Schauen wir uns das anhand eines Beispiels an:

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

In diesem Beispiel haben wir keine `using namespace std;` Direktive verwendet. Auf das Objekt `cout` aus dem Namensraum `std` wird mittels direkter Qualifizierung zugegriffen. Aus dem Operator `<<` wird vom Compiler folgender äquivalenter Code erzeugt:

```
1  operator<<(std::cout, "Hello World");
```

Eigentlich ist der Operator `<<`, den wir verwenden wollen, ebenfalls im Namensraum `std` enthalten. Der Compiler kann allerdings nicht wissen, welchen Operator wir verwenden wollen. Trotzdem funktioniert das Beispiel wie wir uns das vorgestellt haben, da der Compiler sich das Argument `std::cout` ansieht und auf Grund der Zugehörigkeit von `cout` zu `std` den Operatorfunktionsaufruf ebenfalls im Namensraum `std` sucht und auch findet.

Die genauen Regeln für ADL sind etwas komplizierter. Im Standard wird von *associated namespaces* gesprochen, die herangezogen werden, um nach der Funktion zu suchen:

- Wenn der Parameter ein Mitglied einer Klasse ist, dann besteht der Namensraum aus der Klasse selbst und auch aus den Namensräumen in dem die Klasse enthalten ist.
- Wenn der Parameter ein Mitglied eines Namensraumes ist, dann werden die umschließenden Namensräume herangezogen.
- Wenn der Datentyp des Parameters ein eingebauter Datentyp ist, dann gibt es keinen Namensraum in dem zusätzlich gesucht wird.

ADL ist extrem praktisch, da es eine Menge an Tipparbeit erspart, aber es kann auch teilweise zu unerwarteten Ergebnissen kommen. Es ist wichtig zu wissen, dass alle über ADL ermittelten Funktionen prinzipiell gleichwertig sind und auf Grund des Überladens durchaus auch eine Funktion aus einem anderen Namensraum gewählt wird, als man es unter Umständen erwartet:

```
1  // adl.cpp
2  #include <iostream>
3  #include <vector>
4
5  using namespace std;
6
7  namespace MathUtils {
8      class IntVector {};
9      int size(IntVector, int) {
10         return 1;
11     }
12 }
13
14 namespace GameEngine {
15     MathUtils::IntVector v;
16
17     int size(MathUtils::IntVector, unsigned) {
18         return 2;
19     }
20
21     int calculate() {
22         return size(v, 10);
23     }
24 }
25
26 int main() {
27     cout << GameEngine::calculate() << endl;
28 }
```

Bei der leeren Klasse `IntVector` handelt es sich um die Definition eines benutzerdefinierten Datentyps. Wir werden uns benutzerdefinierte Datentypen im Abschnitt **8.1** noch ansehen.

Wahrscheinlich erwartet man sich, dass dieses Beispiel `2` ausgeben wird, nicht wahr? In Wirklichkeit wird aber `1` ausgegeben werden!

Es ist zwar so, dass es eine Definition `int size(MathUtils::IntVector, unsigned)` in diesem Namensraum gibt, jedoch wird diese in diesem speziellen Fall nicht verwendet. Das liegt daran, dass diese nicht die einzige Definition ist, die zur Auswahl herangezogen wird. Es exisitiert jedoch auch noch die Definition `int size(MathUtils::IntVector, int)` im globalen Namensraum, wodurch es zu einer Überladung kommt. Da der aktuelle Parameter den Typ `int` hat, wird die Definition im globalen Namensraum verwendet.

Wäre die Variante mit dem `unsigned` Parameter die einzige Definition, dann würde die ganze Zahl `10` vom Typ `int` auch implizit in einen `unsigned` konvertiert werden und es würde diese Funktion des Namensraum `GameEngine` ausgeführt werden.

Da es aber über die ADL noch eine Funktion `size` gibt, die verwendet werden kann, sieht die Sache anders aus. Es gibt zwei Funktionen, die gleich heißen und sich durch die Typen der Parameter unterscheiden. Also liegt eine Form der Funktionsüberladung vor. Da die Funktion `size` im Namensraum `MathUtils` in diesem Fall keine Konvertierung der Argumente erfordert, wird diese für den Funktionsaufruf herangezogen.

7.4.4 Unbenannte Namensräume

Es gibt noch eine weitere Form der Definition eines Namensraumes, indem der Namensraum selbst keinen Namen erhält. Der Zweck von unbenannten Namensräumen liegt darin, dass alle enthaltenen Bezeichner automatisch im globalen Namensraum für diese Übersetzungseinheit sind. Allerdings stehen diese *nicht* zum externen Linken zur Verfügung, da diese Bezeichner interne Bindung aufweisen.

Man kann sich das so vorstellen, dass ein unbenannter Namensraum äquivalent zu folgendem Konstrukt ist:

```
namespace unique_for_scope {
    /* ... */
}
using unique_for_scope;
```

Das Besondere daran ist, dass der Bezeichner vom Compiler eindeutig gewählt wird und es daher keinen anderen gleichen Bezeichner gibt.

Das folgende Beispiel – bei dem es sich um keinen guten Programmierstil handelt – demonstriert auf einfache Weise die Handhabung von unbenannten Namensräumen.

```
// namespace_unnamed.cpp
#include <iostream>
#include <vector>

using namespace std;

namespace {
    int sum;
    void count(vector<int> v) {
        for (auto i : v)
            sum += i;
    }
}

int main() {
    count({1, 2, 3, 4, 5});
    cout << sum << endl;
}
```

7.5 Headerdateien

Aus den Überlegungen des Kapitels ergibt sich, dass nur gewisse Deklarationen in Headerdateien sinnvoll sind und deshalb in Headerdateien vorkommen dürfen:

- Typaliase, also zum Beispiel:

```
1  using IntStack = std::vector<int>;
```

- Funktionsdeklarationen, aber keine Funktionsdefinitionen:

```
1  void push(IntStack, int);
```

- `inline`-Funktionsdefinitionen, wie zum Beispiel:

```
1  inline
2  double max(double a, double b) { return (a > b) ? a : b; }
```

- `constexpr`-Funktionsdefinitionen:

```
1  constexpr double squared(double x) { return x * x; }
```

- Definitionen von Konstanten, also mit `const` oder mit `constexpr`:

```
1  const double pi{atan(1) * 4};
2  constexpr double area2{squared(2) * pi};
```

- Variablendeklarationen, die keine Definitionen sind, wie zum Beispiel:

```
1  extern int errno;
```

- Namensräume entweder in einer benannten Form oder in einer inline-Variante, jedoch nicht als unbenannte Namensräume! Hier folgt ein benannter Namensraum:

```
1 namespace math {
2 }
```

- Deklarationen und Definitionen von Klassen, Strukturen und Aufzählungen (siehe Abschnitt **8.1**)
- Template-Deklarationen und Template-Definitionen (siehe Abschnitt **13.1**)

Gewisse syntaktische Elemente sollten *niemals* in einer Headerdatei vorkommen:

- Gewöhnliche Funktionsdefinitionen, also solche, die weder `inline` noch `constexpr` und auch keine Templatefunktion sind, da es sonst mehrfache Definitionen dieser Funktion geben würde. Das würde zu Fehlern beim Linken führen.
- Variablendefinitionen dürfen ebenfalls nicht vorkommen, da es sonst zu mehrfachen Vorkommen von Variablen und damit wieder zu Fehlern beim Linken kommen würde.
- Unbenannte Namensräume, da diese zwar die Bezeichner im globalen Namensraum der aktuellen Übersetzungseinheit einpflanzen, aber diese interne Bindung aufweisen. Es ist der Sinn und Zweck von unbenannten Namensräumen den Zugriff zu beschränken und haben sie in Headerdateien nichts verloren.
- `using` Direktiven, wie zum Beispiel `using namespace std;`, da diese die Gefahr von Namenskollisionen stark erhöhen und zu unübersichtlichen Programmen führen, da man nur schwer eruieren kann, woher ein Bezeichner kommt.

8 Klassen

Die grundlegende Idee der Objektorientierung ist, dass Daten und zugehörige Operationen in einer Einheit gruppiert werden. Dazu steht als Hilfsmittel das bekannte Konzept der *Klasse* zur Verfügung, das den Aufbau und das Verhalten gleichartiger Objekte beschreibt.

C++ bietet vielfältige Möglichkeiten, um Klassen zu beschreiben.

8.1 Deklaration und Definiton einer Klasse

In diesem Abschnitt werden wir uns mit den grundlegenden Möglichkeiten beschäftigen, wie man in C++ Klassen deklarieren und definieren kann.

Wir haben schon gesehen, dass mittels `struct` eine Klasse definiert wird, deren Attribute und Methoden alle öffentlich zur Verfügung stehen.

Ein Attribut wird in C++ als *data member* (dt. so viel wie Datenmitglied) und eine Methode wird als *member function* (dt. so viel wie Mitgliedsfunktion) bezeichnet.

Schauen wir uns zuerst ein Beispiel an, das eine Person definiert, die vorerst nur über zwei Attribute verfügt:

```
1  // class.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  struct Person {
7      string name;
8      double weight;
9  };
10
11 int main() {
12     Person p;
13     p.name = "Max";
14     p.weight = 79.5;
15     Person q;
16     cout << "Name: " << p.name << ", Gewicht: "
17          << p.weight << endl;
18     cout << "Name: " << q.name << ", Gewicht: "
19          << q.weight << endl;
20 }
```

Die Ausgabe dieses Programmes könnte folgendermaßen aussehen:

```
1  Name: Max, Gewicht: 79.5
2  Name: , Gewicht: 2.18846e-314
```

Aus diesem Quelltext und der Ausgabe lässt sich einiges ablesen:

- Wie schon gesagt, stehen alle Attribute eines `struct`s öffentlich zur Verfügung. Das bedeutet, dass von außerhalb der Klasse darauf zugegriffen werden kann, wie dies beim Zugriff auf die Attribute `name` und `weight` in der Funktion `main` zu sehen ist.
- Da es sich bei `p` und `q` um Objekte des Typs `Person` handelt, haben beide unterschiedliche, eigene Werte für die Attribute.
- Weiters erkennen wir, dass weder `name` noch `weight` beim Anlegen initialisiert werden. Lediglich im Fall von `p` werden den beiden Attribute nachträglich

Werte zugewiesen. Bei der Ausgabe der uninitialisierten Attribute von `q` sehen wir, dass bei der Ausgabe von `name` ein Leerstring ausgegeben wird, während bei der Ausgabe der Größe ein anscheinend beliebiger Wert aufscheint.

Daraus können wir ablesen, dass Attribute eines benutzerdefinierten Typs initialisiert werden, Attribute eines eingebauten Typs jedoch nicht!

Man kann allerdings auch direkt in der Deklaration der Klasse die Attribute initialisieren. Ändere dazu die Deklaration des Attributs `weight` wie folgt ab, sodass das Gewicht mit `0.0` initialisiert wird:

```
1  double weight{};
```

Danach wird die Ausgabe folgendermaßen aussehen:

```
1  Name: Max, Gewicht: 79.5
2  Name: , Gewicht: 0
```

Will man die den Namen und das Gewicht beim Anlegen der Person angeben und somit die nachträgliche Zuweisung vermeiden, dann benötigt man einen Konstruktor:

```
1  struct Person {
2      Person(string name, double weight) {
3          this->name = name;
4          this->weight = weight;
5      }
6      string name;
7      double weight{};
8  };
```

Das Anlegen der Instanzen soll jetzt folgendermaßen aussehen:

```
1  Person p{"Max", 79.5};
2  Person q("", 0);
```

Damit funktioniert das Programm jetzt wieder wie gewohnt und wir können uns den Details zuwenden:

- In C++ ist ein Konstruktor also eine Methode, die den Namen der Klasse trägt und keinen Rückgabetyp aufweist. Dieser Konstruktor wird ausgeführt, wenn ein Objekt dieser Klasse angelegt wird. Daher wird ein Konstruktor zum Initializieren des Objektes verwendet.

- Der Rumpf des Konstruktors wurde direkt innerhalb der Klassendeklaration angeführt. Das hat die gleiche Bedeutung als wenn wir eine Funktion mit dem Spezifizierungssymbol `inline` markiert hätten. Wir weisen den Compiler an, den Rumpf dieses Konstruktors wenn möglich direkt in den Code einzufügen.

- Innerhalb des Rumpfes sehen wir, dass wir einen Zeiger `this` verwenden. Jede Methode eines Objektes verfügt implizit über so einen Zeiger, der direkt auf das Objekt selber verweist. In diesem konkreten Fall wurde diese Konstruktion verwendet, um die Namen der Parameter gleich benennen zu können wie die Namen der Instanzvariablen.

  Innerhalb des Rumpfes wurde mit der Angabe der Parameter, die Bezeichner dieser Parameter in den Gültigkeitsbereich des Rumpfes des Konstruktors übernommen. Mittels `this->` steht eine Möglichkeit zur Verfügung, auf die Instanzvariablen zuzugreifen.

  Ohne Verwendung von `this` funktioniert unser Beispiel nicht wie wir uns dies erwarten:

```
1  Person(string name, double weight) {
2      name = name;
3      weight = weight;
4  }
```

  Der Compiler wird das Beispiel ohne Fehler übersetzen, aber es wird verständlicherweise nicht wie erwartet funktionieren, da die Parameterdeklaration die Deklaration der Instanzvariablen überschattet. So erhält man in diesem Fall lediglich, dass die Parameter sich selbst die eigenen Werte zuweisen. Die Instanzvariablen werden nicht gesetzt.

  Alternativ hätten wir den auch Konstruktor folgendermaßen definieren können:

```
1  Person(string name_, double weight_) {
2      name = name_;
3      weight = weight_;
4  }
```

Damit lauten die Bezeichner der Parameter anders als die Bezeichner der Instanzvariable. Aus diesem Grund können die Bezeichner der Instanzvariablen ohne `this->` verwendet werden. Das hat allerdings rein syntaxtische Auswirkungen, da der Compiler – von der Bedeutung her – immer ein `this->` voranstellt:

```
1  Person(string name_, double weight_) {
2      this->name = name_;
3      this->weight = weight_;
4  }
```

- Die angegebenen Parameter werden beim Anlegen eines Objektes mitgegeben. Wir sehen, dass wir die Initialisierung sowohl in der Form der vereinheitlichten Initialisierung als auch in der Form eines Funktionsaufrufes vornehmen können.

  Wie schon besprochen haben, ist es sinnvoller die vereinheitlichte Form zu wählen.

- Weiters sehen wir, dass wir jetzt auch die Variable `q` mit Hilfe dieses Konstruktors initialisieren müssen. Das lässt sich leicht verifizieren, indem die ursprüngliche Deklaration in der Form von `Person q;` wieder eingesetzt wird. Der Compiler wird dies zurückweisen, da kein Konstruktor zur Verfügung steht, der keine Parameter erwartet.

  Jetzt stellt sich die Frage, wie dies ohne Konstruktor funktionieren konnte. Das liegt daran, dass der Compiler einen Default-Konstruktor generiert, wenn es keinen anderen Konstruktor gibt.

  Ein generierter Default-Konstruktor hat keine Parameter und initialisiert die Instanzvariablen gemäß den Regeln, die wir schon in diesem Abschnitt kennengelernt haben.

  Natürlich steht es uns frei einen Default-Konstruktor selber zu schreiben und damit `Person q` wieder wie gewohnt verwenden zu können:

```
1  Person() {}
```

In unserem konkreten Fall geht dies in dieser einfachen Art und Weise, da wir direkt bei der Deklaration der Instanzvariablen die gewünschte Initialisierung vorgenommen haben.

Alternativ könnte man Default-Parameter zu dem Konstruktor mit den Parametern hinzufügen, wodurch ebenfalls ein Default-Konstruktor entsteht, da dieser Konstruktor jetzt

```
1  Person(string name="", double weight=0) {
2      this->name = name;
3      this->weight = weight;
4  }
```

Damit besteht die Möglichkeit, die Initialisierungen direkt bei der Instanzvariable `weight` zu entfernen:

```
1  double weight;
```

Allerdings behebt dies nicht das Problem, dass die Variable `name` zuerst mit einem Leerstring initialisiert wird und danach mit dem übergebenen Parameter im Konstruktor überschrieben wird (durch eine Zuweisung). Diesen Umstand kann mit folgender Syntax zu Leibe gerückt werden:

```
1  Person(string name="", double weight=0) : name{name},
2    weight{weight} { }
```

Damit wird die Instanzvariable `name` mit dem Parameter `name` und die Instanzvariable `weight` mit dem Parameter `weight` initialisiert. Eine weitere Initialisierung `name` beziehungsweise `weight` findet nicht statt.

Eine Objekt kann auch default-mäßig mittels einer Kopie eines Objektes gleichen Typs initialisiert werden. Dazu stellt der Compiler diese Funktionalität selbständig in der Art zur Verfügung, dass jede Instanzvariable kopiert wird:

```
1  Person r{p};
2  cout << "Name: " << r.name << ", Gewicht: " << r.weight << endl;
```

Die Ausgabe wird wie erwartet wie die Ausgabe von `p` sein.

Auf die gleiche Art und Weise besteht default-mäßig die Möglichkeit ein Objekt einem anderen Objekt zuzuweisen. Auch hierfür stellt der Compiler die Funktionalität zur Verfügung, sodass jede Instanzvariable kopiert wird:

```
1  r = q;
2  cout << "Name: " << r.name << ", Gewicht: " << r.weight << endl;
```

Damit ergibt sich die Ausgabe wie bei der Ausgabe von `q`.

Im Sinne der Datenkapselung (engl. *encapsulation*) macht es keinen Sinn, alle Attribute und Methoden öffentlich verfügbar zu machen.

Ändere deshalb das `struct` wie folgt ab:

```
1  struct Person {
2      Person(string name="", double weight=0) : name{name},
3        weight{weight} { }
4    private:
5      string name;
6      double weight;
7  };
```

In dieser Form lässt sich das Programm jedoch nicht mehr übersetzen! Das Hinzufügen von `private:` bedeutet, dass die folgenden Bezeichner nicht mehr von außerhalb der Klasse aus zugreifbar sind. Damit kann `p.name` nicht mehr funktionieren.

Wie ist vorzugehen? Jetzt ist es Zeit, Methoden einzuführen, die öffentlich zugreifbar sind und die jeweilige Instanzvariable zurückliefern. Solche Methoden werden oft als „getter method“ bezeichnet:

```
1  struct Person {
2      Person(string name="", double weight=0) : name{name},
3        weight{weight} { }
4      string get_name() { return name; }
5      double get_weight() { return weight; }
6    private:
7      string name;
8      double weight;
9  };
```

Natürlich muss jetzt auch noch der Zugriff der Instanzvariablen in `main` auf den Aufruf der Methode abgeändert werden:

```
1  cout << "Name: " << p.get_name() << ", Gewicht: "
2       << p.get_weight() << endl;
3  cout << "Name: " << q.get_name() << ", Gewicht: "
4       << q.get_weight() << endl;
```

Da wir die Instanzvariablen mittels `private` verborgen haben, müssen wir den Zugriff über die Methoden erfolgen lassen.

Jetzt macht es Sinn, das Schlüsselwort `struct` in `class` umzuändern. Im Unterschied zu `struct`, ist die Bedeutung von `class`, dass die Standardsichtbarkeit `private` ist. Das zieht allerdings nach sich, dass die Sichtbarkeit der Methoden mittels `public` sichergestellt werden muss, damit von `main` aus, diese zugegriffen werden kann:

```
1  class Person {
2      string name;
3      double weight;
4    public:
5      Person(string name="", double weight=0) : name{name},
6        weight{weight} { }
7      string get_name() { return name; }
8      double get_weight() { return weight; }
9  };
```

Da die Standardsichtbarkeit von `class` eben `private` ist, wurde die Deklaration der Instanzvariablen nach vor geschoben. Allerdings ist es übersichtlicher, die alte Reihenfolge wieder herzustellen, sodass die öffentliche Schnittstelle der Klasse wieder in den Vordergrund rückt:

```
1  class Person {
2    public:
3      Person(string name="", double weight=0) : name{name},
4        weight{weight} { }
5      string get_name() { return name; }
6      double get_weight() { return weight; }
7    private:
8      string name;
9      double weight;
10 };
```

Wir sehen hier, dass der öffentliche Teil der Klasse mit den Methoden am Anfang steht und die Daten, die im Sinne der Kapselung von außen nicht sichtbar und zugreifbar sein sollten, im privaten Teil am Ende angeordnet sind. Damit muss zwar ein zusätzliches `private:` angeführt werden, aber die Übersichtlichkeit konnte gesteigert werden.

Will man die Instanzvariablen verändern, dann werden Methoden notwendig, die die Veränderung vornehmen. So eine Methode wird meist als „setter method“ bezeichnet:

```
1  void set_name(string name) { this->name = name; }
2  void set_weight(double weight) { if (weight > 0) this->weight = weight; }
```

Als Alternative für diese Notation mittels `set_` und `get_` bietet es sich auch an, auf überladene Methoden zurückzugreifen:

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Person {
6    public:
7      Person(string name="", double weight=0) : name_{name}, weight_{weight} {
8      string name(string name="") { if (name != "") name_ = name; return name_
9      double weight(double weight=0) { if (weight > 0) weight_ = weight; retur
10   private:
11     string name_;
12     double weight_;
13 };
14
15 int main() {
16     Person p{"Max", 79.5};
17     Person q;
18     cout << "Name: " << p.name() << ", Gewicht: " << p.weight() << endl;
19     q.name("Mini");
20     q.weight(60);
21     cout << "Name: " << q.name() << ", Gewicht: " << q.weight() << endl;
22 }
```

Als Vorteil lässt sich anführen, dass die Anzahl der Methoden reduziert werden konnte und damit, aus einer gewissen Sichtweise, die Bedienung der Klasse übersichtlicher wurde. Der Nachteil ergibt sich durch die notwendige Abfrage im Rumpf der Methode.

8.2 Methoden

In diesem Abschnitt werden wir näher auf die verschiedenen Möglichkeiten im Zusammenhang mit der Deklaration von Methoden eingehen. Methoden werden in C++, wie schon erwähnt, als „member function“ bezeichnet.

8.2.1 Methoden in einer .cpp Datei

Methoden können entweder direkt in einer Headerdatei definiert werden (siehe Abschnitt 243) oder außerhalb in einer .cpp Datei.

Hier betrachten wir den Fall, dass wir die Definition in einer .cpp Datei vornehmen. In der Headerdatei befindet sich in der Klasse lediglich die Deklaration der Methode:

```
1  // car.h
2  #ifndef CAR_H
3  #define CAR_H
4
5  #include <iostream>
6
7  class Car {
8    public:
9      void drive();
10 };
11 #endif
```

Die Definition erfolgt in der zugehörigen Headerdatei:

```
1  // car.cpp
2  #include "car.h"
3
4  void Car::drive() {
5      std::cout << "now driving..." << std::endl;
6  }
```

Beachte, dass der Klassenname gemeinsam mit dem :: Operator vor den Methodennamen gesetzt wird, um dem Compiler mitzuteilen zu welcher Klasse die Methode gehört.

Die Verwendung dieser Klasse findet typischerweise in einer anderen Datei statt:

```
1  // methods.cpp
2  #include <iostream>
3
4  #include "car.h"
5
6  using namespace std;
7
8  int main() {
9      Car car;
10     car.drive();
11 }
```

Bei der Übersetzung muss natürlich beachtet werden, dass das Modul `car.cpp` zuerst übersetzt werden muss und danach entsprechend gelinkt wird.

Im Abschnitt **231** haben wir schon gesehen, dass `this` einen Zeiger auf das aktuelle Objekt darstellt, der implizit immer vorhanden ist, um auf die Instanzvariablen zuzugreifen, wenn sich diese mit den Parametern überschneiden.

`this` kann allerdings auch als Rückgabewert verwendet werden:

```
1  // methodchaining.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  class Adder {
7    public:
8      Adder& add(int val) { amount += val; return *this; }
9      int value() { return amount; }
10   private:
11     int amount{};
12 };
13
14 int main() {
15     Adder a;
16     a.add(1).add(2).add(3);
17     cout << a.value() << endl;
18 }
```

Diese Art der Verarbeitung wird als Methodenverkettung (engl. *method chaining*) zeichnet.

8.2.2 `inline`-Methoden

Wir haben schon gesehen, dass Methoden, die in der Klasse direkt definiert werden, vom Compiler so betrachtet werden, als wenn man diese mit `inline` gekennzeichnet hätte.

Wie auch normale Funktionen können Methoden explizit mit `inline` gekennzeichnet werden. In diesem Fall muss die Definition der Methode natürlich in der Headerdatei stehen und nicht in der `.cpp` Datei:

```
1  // inline_car.h
2  #ifndef INLINE_CAR_H
3  #define INLINE_CAR_H
4
5  #include <iostream>
6
7  class Car {
8    public:
9      void drive();
10 };
11
12 inline void Car::drive() {
13     std::cout << "now driving..." << std::endl;
14 }
15
16 #endif
```

Die Verwendung unterscheidet sich klarerweise nicht von der nicht-`inline` Variante. Lediglich das zusätzliche Binden entfällt.

8.2.3 `const`-Methoden

Methoden können auch als `const` markiert werden, wobei die Bedeutung darin liegt, dass auf konstante Objekte nur `const`-Methoden, nicht jedoch nicht-`const`-Methoden, aufgerufen werden können. Schauen wir uns das an Hand eines Beispiels an, sodass die Anwendung einsichtig wird.

Ändere bitte die Definition von `value()` so ab, dass `const` hinzugefügt wird:

```
1  int value() const { return amount; }
```

Damit wird der Compiler die Datei wie gewohnt übersetzen und das Ergebnis der Ausführung dieses Programmes wie vorher sein. Als nächsten Schritt werden wir eine konstante Variable vom Typ `Adder` anlegen und sonst wie vorher verfahren:

```
1  const Adder b;
2  b.add(1).add(2).add(3);
3  cout << b.value() << endl;
```

Der Compiler wird diesen Programmtext jetzt mit einer Fehlermeldung zurückweisen, da die Methode `add` nicht als `const` Methode deklariert ist und hiermit es zu einer Veränderung einer als `const` deklarierten Variable kommen würde.

Das Programm übersetzt nur, wenn die Methode `add()` für das Objekt `b` entfernt wird, womit lediglich der Wert 0 ausgegeben wird. Damit dieses Beispiel Sinn macht, muss man natürlich einen Konstruktor hinzufügen:

```
1  Adder(int val=0) : amount{val} {}
```

Mit einer Initialisierung der Variablen erhält man eine Konstante, die in bestimmten Situationen sehr praktisch sein kann:

```
1  const Adder b{3};
2  cout << b.value() << endl;
```

Es gibt Situationen in denen es angebracht ist, ein Objekt als `const` zu markieren, aber gewisse Instanzvariablen sehr wohl verändert werden sollen. Das werden wir im Abschnitt **249** betrachten.

Die Vorteile solcher mit `const` markierten Methoden sind:

- Der Compiler kann eine entsprechende Überprüfung vornehmen und damit sind unabsichtliche Veränderungen eines Objektes nicht mehr möglich.
- Außerdem dient diese Markierung auch als Dokumentation gegenüber dem Benutzer der Klasse.

8.2.4 `static`-Methoden

`static`-Methoden (engl. *static method* oder *static member function*) sind Methoden, die unabhängig von einer konkreten Instanz einer Klasse aufgerufen werden können. Daher werden diese Methoden auch Klassenmethoden genannt.

Das Entwurfsmuster „Factory method“ behandelt das Anlegen eines Objektes, dessen Typ im vorhinein nicht bekannt ist. Die Idee ist, eine Methode zu schreiben, die aufgrund eines Parameters die richtige Instanz anlegt und zurückliefert.

Nehmen wir an, dass wir eine Klasse `Car` schreiben wollen, die eine allgemeine Beschreibung eines Autos darstellen soll. An konkreten Automarken soll es „Andi“ und „Zitrone“ geben und in Abhängigkeit einer Benutzereingabe ein Auto der entsprechenden Automarke erzeugt werden:

```
1  // staticmembermethod.cpp
2  #include <iostream>
3
4  using namespace std;
5
6  class Car {
7    public:
8      static Car* make_car(string type);
9  };
10
11 class Andi : public Car {
12 };
13
14 class  : public Car {
15 };
```

In diesem Beispiel sehen wir bis jetzt schon einige interessante Aspekte:

- In der Klasse `Car` gibt es eine statische Methode (*static member function*), die als Parameter den Autotypnamen erhält und einen Zeiger auf ein konkretes Autoobjekt zurückliefert.
- Weiters gibt es die Klasse `Andi`, die von `Car` abgeleitet ist und keine weiteren Attribute oder Methoden definiert. Abgeleitete Klassen, also Vererbung, werden wir uns im Abschnitt **261** ansehen.

Als nächstes werden wir die statische Methode `make_car` implementieren:

```
1  Car* Car::make_car(string type) {
2      if (type == "Audi")
3          return new Audi();
4      else if (type == "Skode")
5          return new Skoda();
6      else
7          return nullptr;
8  }
```

Diese Funktion ist von der Funktionsweise einfach und trotzdem sind wiederum zwei Aspekte interessant:

- Beachte, dass kein `static` in der Definition dieser Methode vorkommt.
- Es wird das neue konkrete Auto am Heap angelegt. Wer die Verantwortung über dieses neu angelegte Objekt übernimmt, muss in der Dokumentation genau festgelegt werden.

Die eigentliche Verwendung dieser Methode ist einfach:

```
1  int main() {
2      string type;
3      cout << "Autotype: ";
4      cin >> type;
5      Car* car{Car::make_car(type)};
6  }
```

Hier siehst du, dass es kein Objekt der Klasse `Car` gibt und trotzdem die Methode `make_car` aufgerufen werden kann, da diese `static` deklariert ist. Beachte weiter, dass mittels des Bereichsauflösungsoperators auf die Methode zugegriffen wird.

8.3 Datenmember

8.3.1 `static`-Klassenvariablen

Analog zu statischen Methoden ist es statische Variablen, die unabhängig von bereits angelegten Objekten immer zur Verfügung stehen. Damit kann sich die

Klasse selbst einen Zustand abspeichern, der unabhängig von einzelnen Instanzen ist. Solche Variablen werden auch als Klassenvariablen bezeichnet.

Betrachten wir das nachfolgende Beispiel einer Klasse, die für jedes neue Objekt eine neue fortlaufende Nummer, also so etwas wie eine Seriennummer, vergibt:

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Car {
6    public:
7      Car() : sernr_{maxsernr} { maxsernr++; }
8      int sernr() { return sernr_; }
9    private:
10     static int maxsernr;
11     int sernr_;
12 };
13
14 int Car::maxsernr{1};
15
16 int main() {
17     Car c1;
18     Car c2;
19     Car c3;
20     cout << "c1: " << c1.sernr() << endl;
21     cout << "c2: " << c2.sernr() << endl;
22     cout << "c3: " << c3.sernr() << endl;
23 }
```

Die Ausgabe wird erwartungsgemäß folgendermaßen aussehen:

```
1  c1: 1
2  c2: 2
3  c3: 3
```

Wichtig ist, dass C++ es nicht erlaubt Klassenvariablen innerhalb der Deklaration der Klasse vorzunehmen. Aus diesem Grund wurde diese, in dem Beispiel, nach der Klassendeklaration vorgenommen.

8.3.2 `mutable`-Instanzvariablen

Im Abschnitt 244 haben wir schon konstante Methoden als Hilfsmittel zur Sicherstellung der Unveränderbarkeit der Instanz kennengelernt. Allerdings ist es manchmal sinnvoll, dass einzelne Instanzvariablen von dieser Unveränderlichkeit ausgenommen sind.

Es gibt prinzipiell zwei Möglichkeiten wie diese Anforderung erfüllt werden kann. Zuerst schauen wir uns eine Technik an, die sich „Memoization“ (das an das man sich erinnert) nennt. Verwendung findet diese Technik überall wo aufwändige Berechnungen durchzuführen sind oder bei dem Zugriff auf externe Ressourcen (Netzwerk, Dateien, Datenbanken,...).

Der Einfachheit halber betrachten wir eine Klasse `Polyline`, die einen Polygonzug darstellen soll. Diese Klasse soll es ermöglichen eine Folge von Punkten des zweidimensionalen Raumes als Polygonzug zu interpretieren. An Operationen sind das Hinzufügen eines weiteren Punktes und die Ermittlung der Länge interessant. Schauen wir uns dazu einmal nur die Schnittstelle an:

```
1  // polyline.cpp
2  #include <iostream>
3  #include <vector>
4  #include <utility>
5  #include <cmath>
6
7  using namespace std;
8
9  class Polyline {
10   public:
11     Polyline(initializer_list<pair<double, double>> points);
12     void add(double x, double y);
13     double length() const;
14 };
```

Wir sehen, dass vorerst alle implementierungsspezifischen Einzelheiten weggelassen wurden.

Die Verwendung soll in weiterer Folge so funktionieren:

```
int main() {
    Polyline poly1{{}};
    cout << poly1.length() << endl;
    poly1.add(0, 0);
    cout << poly1.length() << endl;
    poly1.add(1, 1);
    cout << poly1.length() << endl;
    poly1.add(2, 0);
    cout << poly1.length() << endl;

    const Polyline poly2{{0, 0}, {1, 1}, {2, 0}};
    // poly2.add(3, 3);
    cout << poly2.length() << endl;
}
```

Hier können wir schon sehen wie die Schnittstelle verwendet werden soll:

- Man kann einen neuen Polygonzug mit einer Liste von Punkten anlegen.
- Weiters man kann einen neuen Punkt hinten an den Polygonzug anhängen.
- Außerdem kann man die Länge des Polygonzuges bestimmen.
- Weiters kann ein Polygon konstant sein oder nicht. Wenn ein Polygonzug konstant ist, dann kann kein weiterer Punkt hinzugefügt werden.
- Man kann dies zwar hieraus nicht ablesen, aber als Randbedingung soll gelten, dass die Länge eines Polygonzuges nur berechnet wird, wenn diese benötigt wird.

Jetzt werden wir schrittweise auf die Implementierung eingehen. Starten wir mit den nötigen Instanzvariablen:

```
private:
  vector<pair<double, double>> points;
  mutable double distance{-1};
```

Offensichtlich wird die Liste der Punkte in einem Vektor gespeichert, der Paare von `double`-Werten beinhaltet.

Weiters gibt es eine Instanzvariable `distance`, die die Länge beinhaltet. Da die Länge nur berechnet werden soll, wenn diese benötigt wird, wird in der Distanz ein ungültiger Wert eingetragen. Da eine Distanz nie negativ sein kann, ist damit gekennzeichnet, dass der Wert noch nicht ermittelt worden ist. Die Wirkungsweise von `mutable` werden wir uns später noch ansehen.

Als nächstes werden wir uns die Implementierung des Konstruktors als auch der Methode `add` ansehen:

```
1  Polyline(initializer_list<pair<double, double>> points) : points{points} {}
2  void add(double x, double y) { points.emplace_back(x, y); distance = -1; }
```

Der Konstruktor ist wiederum sehr einfach, da lediglich der Vektor mit den Punkten initialisiert wird. Auch die Implementierung der `add`-Methode ist einfach. Neu ist lediglich die `emplace_back` Methode, die ein neues Paar ohne Zwischenobjekt zum Vektor hinzufügt. Damit wird markiert, dass die Distanz nicht mehr gültig, wird der Wert auf einen ungültigen Wert zurückgesetzt.

Jetzt zur Implementierung der Methode `length`:

```
1  double length() const {
2      if (distance >= 0)
3          return distance;
4      else {
5          const pair<double, double>* p_prev{};
6          for (auto& p : points) {
7              if (!p_prev) {
8                  p_prev = &p;
9                  distance = 0;
10                 continue;
11             }
12             distance += sqrt(pow(p.first - p_prev->first, 2) +
13                              pow(p.second - p_prev->second, 2));
14             p_prev = &p;
15         }
16         return distance;
17     }
18 }
```

Wie funktioniert diese Methode:

- Wenn ein gültiger Wert der Länge schon vorhanden ist, wird dieser zurückgeliefert.
- Anderenfalls muss dieser berechnet werden. Dazu wird über alle Punkte iteriert und jeweils der Abstand zum Vorgänger ermittelt und zur Länge addiert. Dazu wird der Satz von Pythagoras angewandt.
- Lediglich beim ersten Punkt ist kein vorheriger Punkt vorhanden, daher muss dieser Fall extra behandelt werden.

Jetzt sollte das Beispiel wie gefordert funktionieren!

Als Alternative kann zur Verwendung des Schlüsselwort `mutable`, kann man auch das pimpl-Idiom (pointer to implementation) eingesetzt werden. Die Idee ist, die veränderlichen oder besser die gesamte Implementierung der Klasse in ein eigenes Modul auszulagern und in der eigentlichen Klasse nur einen Pointer auf ein Objekt dieser Implementierung zu speichern. Damit benötigt man einerseits kein `mutable` mehr (zumindest in der ursprünglichen Klasse) und andererseits ist in der Schnittstellenklasse keine Implementierung mehr sichtbar!

Das könnte folgendermaßen aussehen:

```
1  // pimpl_polyline.cpp
2  // all necessary includes go here
3
4  class Polyline {
5    public:
6      Polyline(initializer_list<pair<double, double>> points) : impl{new Polyl
7      void add(double x, double y) { impl->add(x, y); }
8      double length() const { return impl->length(); }
9      ~Polyline() { delete impl; }
10   private:
11     PolylineImpl* impl;
12 };
```

Die Klasse `PolylineImpl` wird jetzt in einem eigenem Modul implementiert, das lediglich in kompilierter Form dem API mitgegeben wird.

8.4 Konstruktor und Destruktor

vorgeben und verbieten

Default-Konstruktor, allgemeine Konstruktoren, Typumwandlungskonstruktor Destruktor, delegierender Konstruktor

explicit: Initialisierung mit und ohne `=`

- Initialisierungslisten: Wie schon gesehen: `vector<int> nums(10)` vs. `vector<int> nums{10}`: Die Form mit den geschwungenen Klammern erzeugt eine Initialisierungsliste.
  - Hat der Typ einen Konstruktor mit Initialisierungsliste des entsprechenden Typs, dann wird dieser verwendet.

    Der eingebaute Typ Array wird so betrachtet als hätte dieser einen Konstruktor mit Initialisierungsliste.
  - Wenn der Typ einen Konstruktor mit entsprechender Anzahl und Typ von Parametern hat, dann wird dieser genommen. Unter Umständen werden noch implizite Konvertierungen durchgeführt.

    Auch bei eingebeuten Datentypen wird diese Syntax verwendet. Beachte, dass nur werterhaltende Konvertierungen vorgenommen werden (also keine narrowing Konverterierungen).
  - Hat der Typ keinen Konstruktor, aber öffentlich zugängige Instanzvariable, dann werden diese in der Reihenfolge der Deklaration initialisiert.

Ressourcen mittels RAII (siehe **9.4**).

8.5 Membertypen

8.6 Aufzählungen

enum und enum class

8.7 Forward-Deklarationen

gegenseitige Abhängigkeit von Klassen

9 Ausnahmebehandlung

9.1 Fehlerbehandlungsstrategien

return code, globale Fehlervariable, exception

9.2 Werfen und Abfangen

9.3 `noexcept`-Funktionen

9.4 RAII

10 Einführung in die Standardbibliothek

10.1 Überblick

Die Standardbibliothek von C++ ist sehr umfangreich! In diesem Abschnitt werden wir einen Überblick bekommen und die wesentlichen Prinzipien verstehen werden.

Grob kann man die Standardbibliothek in verschiedene Kategorien gliedern. Hier folgen die Kategorien gruppiert im Überblick:

10.1.1 Sprachunterstützung,

In diese Kategorie fallen Header, die direkt als Ergänzung der Programmiersprache C++ zuzuordnen sind. Wichtige Headerdateien sind:

- `<limits>`: Diese Headerdatei haben wir schon kennengelernt, da diese uns wichtige Informationen über die Implementierung der arithmetischen Datentypen liefert.
- `<cstdint>`: Auch diese Headerdatei kennen wir bereits. Diese stellt weitere Typen für ganze Zahlen mit definierter Größe zur Verfügung.
- `<typeinfo>`: In `<typeinfo>` ist eine gewisse Unterstützung enthalten, um zur Laufzeit Information über die Typen von Speicherobjekten zu ermitteln.
- `<exception>`: Hier sind Funktionen und Typen enthalten, die für das Excpetion-Handling nützlich und notwendig sind.
- `<initializer_list>`: Diese Headerdatei kennen wir ebenfalls schon: Sie stellt den Typ `initializer_list` zur Verfügung.

10.1.2 Utility

Diese Kategorie enthält sinnvolle Funktionen, die oft benötigt werden und allgemeiner Natur sind. Hier gibt es die folgenden wichtigen Headerdateien:

- `<utility>`
- `<tuple>`
- `<bitset>`
- `<memory>`
- `<functional>`
- `<ration>`
- `<chrono>` und `<ctime>`

10.1.3 Mehrsprachigkeit

xxx

10.1.4 Strings

xxx

10.1.5 Ein- und Ausgabe

xxx

10.1.6 Container, Algorithmen, Iterator

xxx

10.1.7 Numerik

xxx

10.1.8 Reguläre Ausdrücke

xxx

10.1.9 Unterstützung für Thread-basierte, nebenläufige Anwendungen

xxx

10.2 Sortieren

Schauen wir uns folgendes Beispiel an, das es erlaubt beliebige Sequenzen zu sortieren, ohne `begin` und `end` verwenden zu müssen:

```
1  // sorting.cpp
2  #include <iostream>
3  #include <algorithm>
4
5  namespace Estd {
6      using namespace std;
7
8      template <typename T>
9      void sort(T& seq) {
10         std::sort(begin(seq), end(seq));
11     }
12 }
13
14 using namespace Estd;
15
16 int main() {
17     int nums[]{3, 5, 2, 1, 4};
18
19     sort(nums);
20
21     for (auto n : nums) {
22         cout << n << ' ';
23     }
24     cout << endl;
25 }
```

10.3 vector

Die schon bekannte Möglichkeit ist, eine Instanz von `vector` zu verwenden, wie wir diese schon kennengelernt haben. Ein `vector` unterscheidet sich von einem rohen Array dadurch, dass es in der Größe veränderbar ist.

Wir wollen wir uns jetzt die Funktionsweise und wichtige Funktionen der Klasse genauer ansehen. Ein `vector` hat eine aktuelle Größe und eine Kapazität, die die Größe des reservierten Speicherbereiches angibt:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v{1, 2};
    cout << "size: " << v.size() << endl;
    cout << "capacity: " << v.capacity() << endl;
}
```

Die Ausgabe könnte folgendermaßen aussehen:

```
size: 2
capacity: 2
```

Da zwei Elemente, nämlich die Zahlen 1 und 2, im Vektor enthalten sind, ist die aktuelle Größe 2. Weiters ist die Kapazität ebenfalls 2.

Hängen wir in weiterer Folge diese Codezeilen an:

```
v.push_back(3);

cout << "push_back(3)" << endl;
cout << "size: " << v.size() << endl;
cout << "capacity: " << v.capacity() << endl;
```

Die Ausgabe sieht bei mir jetzt folgendermaßen aus:

```
size: 2
capacity: 2
push_back(3)
size: 3
capacity: 4
```

Damit sind im Moment die Zahlen 1, 2 und 3 im Vektor enthalten und hiermit beträgt die aktuelle Gräße ebenfalls 3. Nach der Initialisierung mit den beiden Werten 1 und 2 hat der Vektor v nur einen Speicherbereich vom Betriebssystem angefordert, der in der Lage war 2 Elemente aufzunehmen. Dann haben wir ein weiteres Element hinten angehängt und hiermit hat der Vektor einen neuen Speicherbereich anfordern müssen. Offensichtlich hat der Vektor den Speicherbereich doppelt so groß angefordert.

Testen wir weiter durch Anfügen der folgenden Codezeilen:

```
1  v.push_back(4);
2  v.push_back(5);
3  cout << "push_back(4)" << endl;
4  cout << "push_back(4)" << endl;
5  cout << "size: " << v.size() << endl;
6  cout << "capacity: " << v.capacity() << endl;
```

Die zusätzliche Ausgabe sieht bei mir folgendermaßen aus:

```
1  push_back(4)
2  push_back(4)
3  size: 5
4  capacity: 8
```

Hier sehen wir sehr deutlich, dass der Vektor in diesem Schritt wieder einen doppelt so großen Speicherbereich angefordert hat. Prinzipiell hängt die Vorgehensweise von der Implementierung der Standardbibliothek durch den Hersteller zusammen, aber diese Art ist sehr sinnvoll. Wird ein neues Element zum Vektor hinzugefügt, das in den bestehenden Speicherbereich nicht abgelegt werden kann, dann muss der Vektor einen neuen Speicherbereich vom Betriebssystem anfordern, die alten Inhalte in den neuen Speicherbereich kopieren und den alten Speicherbereich dem Betriebssystem zurückgeben.

Damit dieser Vorgang nicht bei jedem Anfügen erfolgen muss, erhöht der Vektor die zusätzliche Größe in der konkreten auf die doppelte Größe.

11 Definieren konkreter Typen

11.1 Überladen von Operatoren

11.2 Kopieren und Verschieben

Kopierkonstruktor, Copy-Assignment, Move-constructor, copy-move

11.3 Typkonversion

- Konstruktor mit einem Argument
- Konversionsoperator

explicit

11.4 Spezielle Operatoren

```
[] () -> ++ -- new delete
```

11.4.1 Definieren eines Funktionsobjektes

11.5 Friend-Klassen

12 Vererbung

12.1 Vererbung in Klassen

12.2 Überladen von Methoden

XXX

A base class and a derived class provide different scopes so that overloading between a base class function and a derived class function doesn't happen by default (p329)!

Siehe Abschnitt **6.3**

```
1 empty buffer
```

13 Templates

13.1 Generelles zu Templates

Deklaration und Definition

13.2 Funktionstemplates

C++ bietet einen sehr mächtigen Mechanismus an, um generische Funktionen und auch Klassen zu entwickeln. In diesem Abschnitt werden wir uns ansehen, wie wir generische Funktionen entwickeln können.

Nehmen wir nochmals unsere Funktion `swap` aus dem Abschnitt **5.3.2** an:

```
1  void swap(string& a, string& b) {
2      string tmp{move(a)};
3      a = move(b);
4      b = move(tmp);
5  }
```

Was ist jetzt wenn, wir eine weitere Funktion wollen, die genau das Gleiche erledigt, nur für ganze Zahlen. Natürlich können wir einfach die obige Funktion kopieren und jedes Vorkommen von `string` durch `int` ersetzen. Damit haben wir eine überladene Funktion geschrieben, die genau dies erledigt. Jetzt könnte es sein, dass wir noch so eine Funktion wollen, die dies für Gleitkommazahlen erledigt. Natürlich können wir wieder mit dem Kopieren und Abändern beginnen und es würde auch funktionieren, aber es macht keinen Sinn! Das kann der Compiler viel besser als wir.

Deshalb wollen wir Schablonen (engl. *templates* oder eingedeutscht Templates) einsetzen:

```
1  // swaptemplate.cpp
2  #include <iostream>
3  #include <memory>
4
5  template <typename T>
6  void swap(T& a, T& b) {
7      T tmp{std::move(a)};
8      a = std::move(b);
9      b = std::move(tmp);
10 }
11
12
13 int main() {
14     int i1{1};
15     int i2{2};
16
17     swap(i1, i2);
18     std::cout << "i1 = " << i1 << ", i2 = " << i2 << std::endl;
19 }
```

Die Ausgabe wird genau das Ergebnis der Funktion `swap` widerspiegeln. XXX

Variable Anzahl an Parameter

13.2.1 Generische Lambdafunktion in Templates

C++14

Generische Lambdafunktionen (siehe Abschntt 6.4.3) können im Zusammenhang mit Templates sinnvoll eingesetzt werden:

XXX ist abzuändern und fertigzustellen

```
template<class C>
void print_elements(const C & c) {
std::for_each(begin(c), end(c), [](auto & x) {
std::cout << x << " ";
});
std::cout << std::endl;
}
```

```
vector<int> v{ 1, 2, 3 };
list<string> w{ "C++", "is", "cool" };

print_elements(v);
print_elements(w);
```

13.3 Variablentemplates

C++11 erlaubt Templates nur für Funktionen, Klassen und Typaliase und C++14 erweitert die Verwendung von Templates auf Variablen:

C++14

```
// vartemplates.cpp
##include <iostream>

using namespace std;

template <typename T>
constexpr T pi = T{3.14159265358979323 85};

template <typename T>
T circle_perimeter(T r) {
return 2 * r * pi<T>;
}

int main() {
cout << circle_perimeter(2.5) << endl;
}
```

A Anhang

A.1 Entwicklungswerkzeuge

Um mit C++ entwickeln zu können, werden Entwicklungswerkzeuge wie Compiler, Linker und Debugger benötigt. Diese sind naturgemäß je Betriebssystem unterschiedlich. Prinzipiell steht die Gnu Compiler Collection sowohl für Windows als auch für Mac OSX und Linux zur Verfügung. Trotzdem macht es durchaus Sinn, eine auf die jeweilige Betriebssystemplattform abgestimmte Variante zu wählen:

- Unter *Windows* wird meist der Compiler Microsoft Visual C++ verwendet, der in der Entwicklungsumgebung Microsoft Visual Studio enthalten ist. Es wird von Microsoft auch in einer kostenlosen Version angeboten.
- Unter *Mac OSX* muss die Entwicklungsumgebung XCode von der Entwicklersite von Apple besorgt und installiert werden. Bei der Installation ist zu beachten, dass die *command line tools* zu installieren sind. Die aktuelle Version enthält den C++ Compiler Clang.
- Unter Linux sind die Entwicklungswerkzeuge ebenfalls getrennt zu installieren. Hier wird meist der C++ Compiler g++ der Gnu Compiler Collection verwendet.

  Für Ubuntu sieht der Befehl zur Installation folgendermaßen aus:

```
1 sudo apt-get install build-essential libgl1-mesa-dev
```

  Verwendest du eine andere oder nicht kompatible Distribution, dann wende dich bitte and die entsprechende Dokumentation.

  Willst du unter Linux mit C++14 programmieren, dann benötigst du g++ mindestens in der Version 4.9, falls diese in deiner Distribution angeboten wird oder du installierst den C++ Compiler Clang. Das kann in Ubuntu folgendermaßen realisiert werden:

```
1  sudo apt-get install clang-3.4 libc++-dev binutils
```

A.2 Übersetzung eines C++ Programmes

Hier geht es nur darum, die Installation zu testen und zu zeigen, wie ein C++ Programm kompiliert wird. Natürlich kann zum Schreiben als auch zum Übersetzen und zum Ausführen eine Entwicklungsumgebung verwendet werden, aber werden wir hier eine Shell benützen.

> Eine Shell (zu Deutsch: Schale) ist ein Programm, das wie eine Schale sich um das Betriebssystem legt und dem Benutzer eine Schnittstelle bietet, um mit dem Betriebssystem per Kommandos zu kommunizieren.
>
> Diese Kommandos werden mittels textbasierter Befehle eingegeben und die Antworten des Betriebssystem werden als Text wieder angezeigt.
>
> Solch eine Shell wird oft auch als Kommandozeileninterpreter bezeichnet, weil es eben ein Programm ist, der Kommandos zeilenweise interpretiert, ausführt und die Ergebnisse zeilenweise anzeigt.
>
> Je Betriebssystem gibt es verschiedene derartige Programme.

Erstelle das klassische „Hello World“ Programm mit einem Texteditor deiner Wahl. Im Abschnitt 2.3 auf der Seite 20 wird der gesamte Quellcode detailliert erklärt:

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, world!" << std::endl;
5  }
```

Speichere diesen Quelltext in eine Datei `hello.cpp`.

Da es sich bei C++ um eine „compiled language“ handelt, muss der Sourcecode zuerst noch in eine ausführbare Datei übersetzt werden, bevor das Programm verwendet werden kann.

Jetzt geht es darum diese Quelldatei in eine ausführbare Datei zu übersetzen und mit den notwendigen Bibliotheken zu linken.

A.2.1 Windows

Öffne eine Shell! Unter Windows kannst du `cmd` oder die PowerShell verwenden.

Wird Visual Studio verwendet, dann sollte der Befehl

```
1 cl /clr hello.cpp
```

in der Shell eingegeben werden können. Damit wird eine ausführbare Datei mit dem Namen `hello.exe` erzeugt, die einfach durch Eingabe von `hello` gestartet werden kann.

Um Warnungen zu sehen, empfehle ich die Option `/Wall` mitzugeben.

Will man eine einzelne Datei lediglich in seine Objektdatei übersetzen, dann ist die Option `/c` (für „compile“) zu verwenden. Die entstehende Datei heißt `hello.obj` (für Objektdatei).

Unter Umständen müssen mehrere Dateien angegeben werden, dann sind diese einfach in der Kommandozeile hintereinander anzugeben.

A.2.2 Mac OSX

Öffne eine Shell! Verwende unter Mac OSX die Applikation `Terminal`, in das die eigentliche Shell läuft.

Aktuelle XCode Versionen verwenden den C++ Compiler clang. Gib den folgenden Befehl in der Shell ein:

```
1 clang++ -stdlib=libc++ -std=c++11 -lc++abi -o hello hello.cpp
```

Die Option `-o` (engl. *output*) bewirkt, dass das lauffähige Programm mit dem Namen `hello` erstellt wird. `-std=c++11` gibt an, dass die Sprachversion C++11 verwendet werden soll. Die restlichen Parameter geben die zu verwendete Bibliothek an.

Anstatt `-std=c++11` kann auch `-std=c++1y` verwendet werden, wenn eine aktuelle Version von `clang` installiert ist. Dies ermöglicht dann auch Features von C++14 zu verwenden.

Auch `clang` kennt die Option `-Wall`, um Warnungen anzuzeigen.

Für die weiteren Beispiele gehe ich davon aus, dass die Umgebungsvariable `PATH` so gesetzt ist, dass das aktuelle Verzeichnis enthalten ist. D.h., dass der Punkt `.` enthalten ist.

Jetzt kann durch Eingabe von `hello` einfach das Programm gestartet werden.

Will man eine einzelne Datei lediglich in seine Objektdatei übersetzen, dann ist die Option `-c` (für „compile“) anzuhängen und die Option `-o hello` nicht anzugeben. Die entstehende Datei heißt `hello.o` (für Objektdatei).

Unter Umständen müssen mehrere Dateien angegeben werden, dann sind diese einfach in der Kommandozeile hintereinander anzugeben.

A.2.3 Linux

Öffne eine Shell! Unter Linux kannst du eines der vielen verschiedenen Terminalprogramme verwenden. Unter Linux läuft innerhalb des Terminalprogrammes die eigentliche Shell.

Für die weiteren Beispiele gehe ich davon aus, dass die Umgebungsvariable `PATH` so gesetzt ist, dass das aktuelle Verzeichnis enthalten ist. D.h., dass der Punkt `.` enthalten ist.

Unter Linux sieht der Befehl folgendermaßen aus:

```
1  g++ hello.cpp -std=c++11 -o hello
```

Die Option `-o` (engl. *output*) bewirkt, dass das lauffähige Programm mit dem Namen `hello` erstellt wird. `-std=c++11` gibt an, dass die Sprachversion C++11 verwendet werden soll.

Anstatt `-std=c++11` kann auch `-std=c++14` oder `-std=c++1y` verwendet werden, wenn eine aktuelle Version von `g++` installiert ist. Dies ermöglicht dann auch Features von C++14 zu verwenden.

Will man sich Warnungen anzeigen lassen, dann empfehle ich die Optionen `-Wconversion` und `-Wall` anzugeben. `g++` hat sehr viele Optionen und diese sollten je Bedarf ausgesucht werden.

Für die weiteren Beispiele gehe ich davon aus, dass die Umgebungsvariable `PATH` so gesetzt ist, dass das aktuelle Verzeichnis enthalten ist. D.h., dass der Punkt `.` enthalten ist.

Jetzt kann durch Eingabe von `hello` einfach das Programm gestartet werden.

Will man eine einzelne Datei lediglich in seine Objektdatei übersetzen, dann ist die Option `-c` (für „compile“) anzuhängen und die Option `-o hello` nicht anzugeben. Die entstehende Datei heißt `hello.o` (für Objektdatei).

Unter Umständen müssen mehrere Dateien angegeben werden, dann sind diese einfach in der Kommandozeile hintereinander anzugeben.

A.3 Zugreifen auf den Exit-Code eines Programmes

Will man auf den Exit-Code eines Programmes zugreifen und weiß nicht wie dann hilft dieser Abschnitt weiter.

Schreibe zuerst den folgenden Code in eine Datei `minimal.cpp` und übersetze dieses Programm in eine ausführbare Datei `minimal`:

```
1  int main() {
2      return 0;
3  }
```

Im folgenden wird erklärt wie auf den Rückgabewert je verwendetem Betriebssystem zugegriffen werden kann.

A.3.1 Windows

Unter Windows gibt es je nach verwendeter Shell verschiedene Möglichkeiten. Wird eine `cmd` Shell verwendet, dann kommt man mit folgendem Befehl ans Ziel:

```
1  minimum
2  echo %ERRORLEVEL%
3  0
```

Danach wird in unserem konkreten Fall der Wert 0 angezeigt.

Wird die PowerShell verwendet, dann sieht der korrekte Befehl folgendermaßen aus:

```
1  minimum
2  echo $LastExitCode
3  0
```

A.3.2 Mac OSX und Linux

Sowohl für Mac OSX als auch für Linux gibt es eine Vielzahl von verschiedenen Shells. Normalerweise kommt eine Shell zum Einsatz, die `bash` heißt. Viele Shells sind bezüglich der Ermittlung des Rückgabewertes kompatibel zu dieser Shell.

Unter Mac OSX und Linux kann man meist auf der Kommandozeile folgendermaßen auf diesen Rückgabewert zugreifen:

```
1  minimum
2  echo $?
3  0
```

Auch hier wird mit unserem Beispiel natürlich wieder 0 angezeigt werden.

Verwendest du die sehr gute Shell `fish`, dann sieht der Befehl folgendermaßen aus:

```
1  minimum
2  echo $status
3  0
```

Solltest du eine andere Shell verwenden, die sich diesbezüglich wiederum anders verhält, dann bist du wahrscheinlich ein Spezialist und benötigst wahrscheinlich keine weitere Hilfe.

A.4 Literaturhinweise

Hier möchte ich sinnvolle Ergänzungen für das weitere Studium in C++ anführen.

Die folgendenden Bücher gelten als „Standard“:

- Die „Bibel“ ist sicher das Buch des Erfinders Bjarne Stroustrup von C++ [Stroustrup, 2013].
- Als das Standardwerk zur Standardbibliothek gilt das Buch [Josuttis, 2012] von Nicolai Josuttis.
- Als Online-Quelle hauptsächlich zur Standardbibliothek empfehle ich [Kanapickas, 2014].

Bist du mehr an einer Einführung interessiert, dann sind die folgenden Werke unter Umständen für dich interessant:

- Ist man mehr an einer eher deutschsprachigen, umfassenden Einführung interessiert, die sich auch an Nicht-Programmierer richtet, dann ist [Wolf, 2014] zu empfehlen.
- Ein ebenfalls gutes deutschsprachiges und ebenfalls umfangreiches Buch, das viele praktische Algorithmen enthält ist [Breymann, 2011].
- Will man eher ein englischsprachige, gute Einführung, dann ist [Paul Deitel, 2014] zu empfehlen.

Hast du schon Erfahrungen mit C++ und möchtest speziell die neuen Features von C++11 kennenlernen oder möchtest diese kompakt in einem Buch nachlesen können,

- dann empfehle ich das Buch [Will, 2012]!

Zu guter letzt kann ich noch zwei ausgezeichnete Bücher empfehlen, die jedoch eindeutig schon in die Kategorie weiterführende Literatur einzuordnen sind:

- Interessiert man sich für parallele Anwendungen, dann empfehle ich [Williams, 2012].
- Ist man eher daran interessiert, wie man gute, wiederverwendbare Programme entwickelt, dann ist [Reddy, 2011] sehr zu empfehlen.

B Glossar

ConTeXt	The best way to write texts

C References

Breymann, U. (2011). *Der C++ Programmierer*. Carl Hanser.
Josuttis, N. M. (2012). *The C++ Standard Library*. Addison-Wesley.
Kanapickas, P. (2014). C++ reference. . Accessed: 2014-08-30.
Paul Deitel, H. D. (2014). *C++11 for Programmers*. Prentice Hall.
Reddy, M. (2011). *API design for C++*. Morgan Kaufmann.
Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley.
Will, T. T. (2012). *C++11 programmieren*. Galileo Press.
Williams, A. (2012). *C++ Concurrency in Action*. Manning Publications.
Wolf, J. (2014). *C++ Das umfassende Handbuch*. Gallileo Press.