Inhaltsverzeichnis

Lizenz

Kapitel 1

Wie lese ich dieses Buch?

Egal, ob Sie schon C++ können, eine andere Programmiersprache beherrschen oder kompletter Programmieranfänger sind. Wenn Sie in C++ programmieren wollen, werden Sie mit diesem Buch etwas anfangen können. Es steht Ihnen frei, das Buch ganz normal von Anfang an zu lesen oder sich die Kapitel herauszupicken, die Sie interessieren. Allerdings sollten Sie beim Herauspicken darauf achten, dass Sie den Stoff der vorherigen Kapitel beherrschen, andernfalls werden Sie wahrscheinlich Schwierigkeiten haben, das Kapitel Ihres Interesses zu verstehen.

Dieses Buch gliedert sich in verschiedene Abschnitte, von denen jeder einen eigenen Schwierigkeitsgrad und entsprechend eine eigene Zielgruppe hat. Die meisten Abschnitte vermitteln Wissen über C++ oder sollen die praktische Anwendung zeigen, um das bisherige Wissen zu festigen. Andere weisen auf Besonderheiten und „Stolpersteine“ hin oder vermitteln Hintergrundwissen.

Am Ende eines jeden Abschnittes steht eine Zusammenfassung, anhand der das gesamte, in diesem Abschnitt beschriebene Wissen, noch mal schnell nachgelesen werden kann. Wer schnell etwas nachschlagen will, kann sich in der Regel an die Zusammenfassungen halten. Auf Abweichungen von dieser Regel wird explizit hingewiesen. Jeder Abschnitt enthält eine Anzahl von Kapiteln, welche einen komplexeren Zusammenhang erklären. Die Abschnitte sind in sich abgeschlossen, setzen aber voraus, dass Sie den Stoff der vorherigen Abschnitte verstanden haben.

Am Ende jedes Kapitels stehen meistens ein paar Fragen und/oder Aufgaben, die Ihnen helfen sollen zu überprüfen, ob Sie verstanden haben, was im entsprechenden Kapitel stand. Die Antworten auf die Fragen lassen sich einfach ausklappen, für die Aufgaben gibt es je einen Verweis auf eine Seite mit einer Musterlösung.

Diese Musterlösung ist natürlich nicht die einzige Variante, um die Aufgabe korrekt zu erfüllen, sie dient lediglich als Beispiel, wie Sie es machen könnten. Zu beachten ist allerdings, dass das im Kapitel erworbene Wissen genutzt werden sollte, um die Aufgabe zu erfüllen.

„Für Programmieranfänger“ ist der nächste Abschnitt. Wenn Sie schon in einer anderen Programmiersprache Programme geschrieben haben, können Sie diesen Abschnitt getrost überspringen. Es geht nicht speziell um C++, sondern eher darum, zu begreifen, was überhaupt eine Programmiersprache ist und wie man dem Rechner sagen kann, was er zu tun hat.

Wenn Sie eine andere Programmiersprache schon sehr sicher beherrschen, reicht es wahrscheinlich, wenn Sie von den nächsten Abschnitten nur die Zusammenfassungen lesen, aber es kann Ihnen in keinem Fall schaden, auch die kompletten Abschnitte zu lesen. Die Zusammenfassungen sollen Ihnen zwar einen groben Überblick geben, was in diesem Abschnitt vermittelt wurde, aber es ist nun einmal nicht möglich, zur gleichen Zeit kurz und vollständig zu sein.

Sofern Sie C++ schon sicher beherrschen, können Sie sich einfach ein Kapitel ausgucken, es lesen und wenn nötig, in einem früheren Kapitel nachschlagen. Umsteigern ist dieses Verfahren hingegen nicht zu empfehlen, weil C++ eben doch etwas anderes ist als die meisten anderen Programmiersprachen.

In jedem Fall hoffen wir, dass dieses Buch Ihnen hilft, Programmieren zu lernen und Ihre Fähigkeiten zu verbessern, aber vergessen Sie niemals, dass ein Buch nur eine Stütze sein kann. *Programmieren lernt man durchs Programmieren!*

Kapitel 2

Es war einmal...

C++ wurde von Bjarne Stroustrup ab 1979 in den AT&T Bell Laboratories entwickelt. Ausgangspunkt waren Untersuchungen des UNIX-Betriebssystemkerns in Bezug auf verteiltes Rechnen.

Auf die Idee für eine neue Programmiersprache war Stroustrup schon durch Erfahrungen mit der Programmiersprache Simula im Rahmen seiner Doktorarbeit an der Cambridge University gekommen. Simula erschien zwar geeignet für den Einsatz in großen Software-Projekten, die Struktur der Sprache erschwerte aber die für viele praktische Anwendungen erforderliche Erzeugung hocheffizienter Programme. Demgegenüber ließen sich effiziente Programme zwar mit der Sprache BCPL schreiben, für große Projekte war BCPL aber wiederum ungeeignet.

Mit den Erfahrungen aus seiner Doktorarbeit erweiterte Stroustrup nun die Programmiersprache C um ein Klassenkonzept, für das die Sprache Simula-67 das primäre Vorbild war. Die Wahl fiel auf die Programmiersprache C, eine Mehrzwecksprache, die schnellen Code produzierte und einfach auf andere Plattformen zu portieren war. Als dem Betriebssystem UNIX beiliegende Sprache hatte C außerdem eine nicht unerhebliche Verbreitung. Zunächst fügte er der Sprache Klassen (mit Datenkapselung) hinzu, dann abgeleitete Klassen, ein strengeres Typsystem, Inline-Funktionen und Standard-Argumente.

Während Stroustrup „C with Classes“ („C mit Klassen“) entwickelte (woraus später C++ wurde), schrieb er auch cfront, einen Compiler, der aus C with Classes zunächst C-Code als Zwischenresultat erzeugte. Die erste kommerzielle Version von cfront erschien im Oktober 1985.

1983 wurde C with Classes in C++ umbenannt. Erweiterungen darin waren: virtuelle Funktionen, Überladen von Funktionsnamen und Operatoren, Referenzen,

Konstanten, änderbare Freispeicherverwaltung und eine verbesserte Typüberprüfung. Die Möglichkeit von Kommentaren, die an das Zeilenende gebunden sind, wurde wieder aus BCPL übernommen (//).

1985 erschien die erste Version von C++, die eine wichtige Referenzversion darstellte, da die Sprache damals noch nicht standardisiert war. 1989 erschien die Version 2.0 von C++. Neu darin waren Mehrfachvererbung, abstrakte Klassen, statische Elementfunktionen, konstante Elementfunktionen und die Erweiterung des Schutzmodells um protected. 1990 erschien das Buch „The Annotated C++ Reference Manual“, das als Grundlage für den darauffolgenden Standardisierungsprozess diente.

Relativ spät wurden der Sprache Templates, Ausnahmen, Namensräume, neuartige Typumwandlungen und boolesche Typen hinzugefügt.

Im Zuge der Weiterentwicklung der Sprache C++ entstand auch eine gegenüber C erweiterte Standardbibliothek. Erste Ergänzung war die Stream-I/O-Bibliothek, die Ersatz für traditionelle C-Funktionen wie zum Beispiel `printf()` und `scanf()` bietet. Eine der wesentlichen Erweiterungen der Standardbibliothek kam später durch die Integration großer Teile der bei Hewlett-Packard entwickelten Standard Template Library (STL) hinzu.

Nach jahrelanger Arbeit wurde schließlich 1998 die endgültige Fassung der Sprache C++ (ISO/IEC 14882:1998) genormt. 2003 wurde ISO/IEC 14882:2003 verabschiedet, eine Nachbesserung der Norm von 1998. Die nächste größere Überarbeitung der Sprache C++ erscheint voraussichtlich im Jahr 2009.

Die vorhandenen Performance-Probleme der Sprache C++, die Rechenzeit und Speicherplatz betreffen, werden auf hohem Niveau zusammen mit Lösungen im Technical Report ISO/IEC TR 18015:2006 diskutiert. Das Dokument ist zum Download von ISO freigegeben.

2.1 Der Name „C++“

Der Name ist eine Wortschöpfung von Rick Mascitti und wurde zum ersten Mal im Dezember 1983 benutzt. Der Name kommt von der Verbindung der Vorgängersprache C und dem Inkrement-Operator „++“, der den Wert einer Variable um eins erhöht.

2.2 Weiterentwicklung der Programmiersprache C++

Um mit den aktuellen Entwicklungen der sich schnell verändernden Computer-Technik Schritt zu halten, aber auch zur Ausbesserung bekannter Schwächen, erarbeitet das C++-Standardisierungskomitee derzeit die nächste größere Revision von C++, die inoffiziell mit C++0x abgekürzt wird, worin die Ziffernfolge eine grobe Einschätzung des möglichen Erscheinungstermins andeuten soll. Im November 2006 wurde der Zieltermin für die Fertigstellung auf das Jahr 2009 festgelegt und die Abkürzung C++09 dafür eingeführt.

Die vorrangigen Ziele für die Weiterentwicklung von C++ sind Verbesserungen im Hinblick auf die Systemprogrammierung sowie zur Erstellung von Programmbibliotheken. Außerdem soll die Erlernbarkeit der Sprache für Anfänger verbessert werden.

2.3 Berücksichtigung aktueller technischer Gegebenheiten

C++ deckt einige typische Problemfelder der Programmierung noch nicht ausreichend ab, zum Beispiel die Unterstützung von Nebenläufigkeit (Threads), deren Integration in C++, insbesondere für die Verwendung in Mehrprozessorumgebungen, die eine Überarbeitung der Sprache unumgänglich macht. Die hierfür nötige Überarbeitung des C++-Speichermodells soll Garantien der Sprache für den nebenläufigen Betrieb festlegen, um Mehrdeutigkeiten in der Abarbeitungsreihenfolge sowohl aufzulösen, als auch in bestimmten Fällen aufrechtzuerhalten und dadurch Spielraum für Optimierungen zu schaffen.

2.3.1 Verbesserungen am Sprachkern

Zu den weiter reichenden Spracherweiterungen gehören die "Konzepte" (engl. concepts) zur Verbesserung der Handhabung von Templates, Typinferenz zur Ableitung von Ergebnistypen aus Ausdrücken und die R-Wert-Referenzen, mit deren Hilfe sich als Ergänzung zu dem bereits vorhandenen Kopieren von Objekten dann auch ein Verschieben realisieren lässt.

2.3.2 Erweiterung der Programmbibliothek

Im April 2006 gab das C++-Standardisierungskomitee den so genannten ersten technischen Report (TR1) heraus, eine nichtnormative Ergänzung zur aktuell gültigen, 1998 definierten Bibliothek, mit der Erweiterungsvorschläge vor einer möglichen Übernahme in die C++-Standardbibliothek auf ihre Praxistauglichkeit hin untersucht werden sollen. Viele Compiler-Hersteller liefern den TR1 mit ihren Produkten aus.

Enthalten sind im TR1 u.a. reguläre Ausdrücke, verschiedene intelligente Zeiger, ungeordnete assoziative Container, eine Zufallszahlenbibliothek, Hilfsmittel für die C++-Metaprogrammierung, Tupel sowie numerische und mathematische Bibliotheken. Die meisten dieser Erweiterungen stammen aus der Boost-Bibliothek, woraus sie mit minimalen Änderungen übernommen wurden. Außerdem sind viele Bibliothekserweiterungen der 1999 überarbeiteten Programmiersprache C (C99) in einer an C++ angepassten Form enthalten.

Mit Ausnahme der numerischen und mathematischen Bibliotheken wurde die Übernahme aller TR1-Erweiterungen in die kommende Sprachnorm C++0x bereits vom C++-Standardisierungskomitee beschlossen. Ein weiterer Bibliothekszusatz namens TR2, mit Funktionalität für den nebenläufigen Betrieb (Threads), ist in Vorbereitung und soll nach dem kommenden Standard veröffentlicht werden.

Kapitel 3

Compiler

Dieses Kapitel ist noch nicht vollständig

Kurz: was ist ein Compiler? Präprozessor? Linker?

3.1 Hilfsmittel

1. Compiler Liste (mit Betriebssystem)
2. IDE Liste (mit Betriebssystem und möglichen Compilern)

3.2 Die GCC

GCC steht für **G**nu **C**ompiler **C**ollection und ist eine Sammlung von freien Compilern. GCC stellt neben dem C++-Compiler g++, noch viele andere Compiler für verschiedene Sprachen bereit. Der C-Compiler ist etwa gcc und der Java-Compiler heißt gcj.

GCC steht unter der GNU General Public License (GNU GPL), das heißt, GCC ist Opensource und jeder ist berechtigt, den Quellcode einzusehen, zu verändern und natürlich auch GCC zu verwenden. Da GCC auf sehr viele Plattformen portiert wurde und unter nahezu allen Betriebssystemen verfügbar ist, eignet er sich prächtig, um ihn in einem Buch wie diesem zu verwenden.

Auf fast allen Linux-, Unix- und Unixartigen-Systemen ist GCC schon standardmäßig installiert. Falls nicht, können Sie ihn mit dem Paketmanager Ihrer (Linux-

)Distribution nachinstallieren oder ihn von [gcc.gnu.org] herunterladen und manuell installieren. Auf Windows können Sie MinGW (ein Portierung von GCC) verwenden. Sie ist zwar nicht so aktuell wie die originale GCC-Version, aber für den Anfang immer noch ausreichend. Alternativ können Sie GCC auf Windows auch über Cygwin nutzen.

Aber jetzt ist Schluss mit der Theorie. Schauen wir uns erst einmal an, wie g++ benutzt wird. Im folgenden wird angenommen, dass eine Datei mit dem Namen *prog.cpp* vorhanden ist. In dieser Datei könnte zum Beispiel folgendes stehen:

```
1. include <iostream>


using namespace std;
int main(){
    cout << "Hallo Welt\n";
}
```

Dieses Programm müssen Sie noch nicht verstehen, (es gibt „Hallo Welt“ aus,) im Kapitel Hallo, du schöne Welt! wird es erklärt. Nun geben Sie auf der Kommandozeile (Linux: Shell, Windows: Eingabeaufforderung) folgendes ein: `g++ prog.cpp`

g++ ist der Name des Programms, welches aufgerufen wird, also der Compiler g++. *prog.cpp* ist der Name der Datei, die kompiliert werden soll. Wenn Sie diesen Befehl ausführt, sehen Sie entweder Fehlermeldungen auf dem Bildschirm, oder aber Sie bekommt eine Datei mit dem Namen a.out. Manchmal bekommen Sie auch sogenannte „warnings“, also Warnungen. Bei diesen Warnungen wird der Code zwar kompiliert, aber Sie sollten versuchen, warnungsfreie Programme zu schreiben. Ein Beispiel für eine Warnung könnte z.B. sein: `g++ prog.cpp`

```
prog.cpp: In function ‘int main()’:
prog.cpp:17: warning: comparison between signed and unsigned integer expressions
```

In diesem Beispiel würde die Warnung bedeuten, dass wir eine Zahl ohne Vorzeichen (`unsigned`) und eine mit Vorzeichen (`signed`) vergleichen, und zwar innerhalb der Funktion ‘`int main()`’, genauer gesagt in Zeile 17. Was `unsigned` und `signed` ist, erfahren Sie im Kapitel Variablen, Konstanten und ihre Datentypen.

Es gibt auch einige, zum Teil hilfreiche Warnungen, die nicht angezeigt werden. Um diese zu sehen, müssen Sie die Option *-Wall* hinzufügen. `g++ -Wall prog.cpp`

Um sich noch mehr Warnungen anzeigen zu lassen (*-Wall* zeigt auch nicht alle), können Sie auch noch *-Wextra* benutzen: `g++ -Wall -Wextra prog.cpp`

Es wird empfohlen, diese Möglichkeit zu nutzen, vor allem wenn Sie auf der Suche nach Fehlern in Ihrem Programm sind.

Möchten Sie, dass das fertige Programm einen anderen Namen als a.out hat, so können Sie es natürlich jedesmal umbenennen. Dies ist aber umständlich und somit nicht zu empfehlen. Ein viel eleganterer Weg ist das benutzen der Option *-o*.

```
g++ -o tollername prog.cpp
```

Der Dateiname nach der Option -o gibt an, in welche Datei das kompilierte Programm gespeichert werden soll. So erhalten Sie in diesem Beispiel die ausführbare Datei *tollername*.

Sollten Sie sich einmal nicht mehr erinnern, was eine bestimmte Funktion bewirkte oder wie sie hieß, so können Sie g++ einfach nur mit der Option *–help* aufrufen.

```
g++ -help
```

g++ gibt dann eine kurze Auflistung der Optionen aus. Wer gerne eine detailliertere Version hätte, kann unter Linux auch das Manualprogramm „man“ benutzen:

```
man g++
```

Allerdings ist diese Hilfe etwas lang (über 10 000 Zeilen...). Außerdem gibt es unter Linux noch eine übersichtliche Dokumentation bei `info`. `info g++`

3.3 Für Fortgeschrittenere

Jetzt sollten Sie erst einmal den Rest des Kapitels überspringen und mit einigen der folgenden Buchkapitel weitermachen, denn für den Rest ist ein wenig Vorwissen sehr hilfreich. Sobald Sie mit den ersten paar Kapiteln fertig sind, können Sie hierher zurückkehren und den Rest lesen.

Was passiert überhaupt, wenn Sie g++ aufrufen? Nun, als erstes wird der Code vom Präprozessor durchgeschaut und bearbeitet. (Natürlich bleibt Ihre Quelldatei, wie sie ist.) Dabei werden beispielsweise Makros ersetzt und Kommentare gelöscht. Dieser bearbeitete Code wird dann vom Compiler in die Assemblersprache übersetzt. Die Assemblersprache ist auch eine Programmiersprache, welche aber nur die Maschinensprache so darstellt, dass ein Mensch sie (leichter) lesen kann. Schließlich wird diese Assemblersprache von einem Assembler in Maschinencode umgewandelt. Zum Schluss wird noch der Linker aufgerufen, der die einzelnen Programmdateien und die benutzten Bibliotheken "verbindet".

Darum, dass dies alles in korrekter Reihenfolge richtig ausgeführt wird, brauchen Sie sich nicht zu kümmern; g++ erledigt das alles für Sie. Allerdings kann es

manchmal nützlich sein, dass das Programm noch nicht gelinkt wird, etwa wenn Sie mehrere Quelldateien haben. Dann kann man einfach dem g++ die Option *-c* mitgeben. `g++ -c -o prog.o prog.cpp`

Durch diesen Aufruf erhalten wir eine Datei *prog.o*, die zwar kompiliert und assembliert, aber noch nicht gelinkt ist. Deswegen wurde die Datei auch *prog.o* genannt, da ein kompiliertes und assembliertes, aber nicht gelinktes Programm als Objektdatei vorliegt. Objektdateien bekommen üblicherweise die Endung *.o. Ohne die -o Option hätten Sie möglicherweise eine Datei gleichen Namens erhalten, aber das ist nicht absolut sicher. Es ist besser den Namen der Ausgabedatei immer mit anzugeben; so können Sie sicher sein, dass die ausgegeben Dateien auch wirklich den von Ihnen erwarteten Namen haben.

Bei g++ gibt es Unmengen von Optionen, mit denen Sie fast alles kontrollieren können. So gibt es natürlich auch eine Option, durch die kompiliert, aber nicht assembliert wird. Diese Option heißt *-S*. `g++ -S prog.cpp`

Diese Option wird allerdings fast nie benötigt, es sei denn Sie interessieren sich dafür, wie Ihr Compiler Ihren Code in Assembler umsetzt. Die Option *-E* ist schon nützlicher. Mit ihr wird nur der Präprozessor ausgeführt: `g++ -E prog.cpp`

So können Sie z. B. sehen, ob mit den Makros alles ordnungsgemäß geklappt hat oder die Headerdateien eventuell in einer von Ihnen unerwarteten Reihenfolge inkludiert wurden. Eine Warnung sollen Sie hier aber mit auf den Weg bekommen. Wenn der Präprozessor durchgelaufen ist, stehen auch alle mittels `#include` eingebundenen Headerdateien der Standardbibliothek mit im Quelltext, die Ausgabe kann also ziemlich lang werden. Allein das Einbinden von `iostream` produziert bei g++ 4.1 knapp 30 000 Zeilen Code. Welche Zeit es in Anspruch nimmt, den Quellcode nach dem Übersetzen in Assembler (Option *-s*) zu lesen, dürfte damit geklärt sein.

Sie sollten auch die Option -ansi kennen. Da g++ einige C++ Erweiterungen beinhaltet, die nicht im C++-Standard definiert sind oder sogar mit ihm im Konflikt stehen, ist es nützlich, diese Option zu verwenden, wenn Sie ausschließlich mit Standard-C++ arbeiten wollen. In der Regel ist das zu empfehlen, da sich solcher Code *viel* leichter auf andere Compiler portieren lässt. Im Idealfall müssten Sie gar keine Änderungen mehr vornehmen. Da aber weder g++ noch irgendein anderer Compiler absolut Standardkonform sind, ist das selten der Fall. Ein Negativbeispiel für standardkonforme Compiler kommt immer mal wieder aus Redmond. Die dort ansässige Firma produziert in der Regel Produkte, die zu ihren selbstdefinierten Standards in einer bestimmten Version (also nicht mal untereinander) kompatibel sind. Beschweren Sie sich bitte nicht, wenn einige Programmbeispiele mit einem Compiler von dort, nicht kompiliert werden. Der Fairness halber soll

trotzdem angemerkt werden, dass langsam aber doch sicher Besserung in Sicht ist. `g++ -ansi -o prog prog.cpp`

Dies bewirkt, dass diese nicht standardkonformen Erweiterungen des g++-Compilers abgeschaltet werden. Solcher Code ist in der Regel die bessere Wahl, da er auch in Zukunft, wenn es neue Compiler oder Compiler-Versionen geben wird, noch mit ihnen übersetzt werden kann.

Möchten Sie eine Warnung erhalten, wenn nicht standardkonforme Erweiterungen von g++ verwendet werden, dann nutzen Sie die Option *-pedantic*: `g++ -pedantic -o prog prog.cpp`

Um die Optimierung zuzuschalten, nutzen Sie die Option *-Ox*, wobei das x für eine Zahl von 1 bis 3 steht. 3 bedeutet stärkste Optimierung. `g++ -O3 -o prog prog.cpp`

Programme, die mit Optimierung übersetzt wurden, sind kleiner und laufen schneller. Der Nachteil besteht in der höheren Zeit, welche für die Übersetzung an sich benötigt wird. Daher sollten Sie die Optimierung nur zuschalten, wenn Sie eine Programmversion erstellen, die Sie auch benutzen möchten. Beim Austesten während der Entwicklung ist es besser, ohne Optimierung zu arbeiten, da häufig kompiliert wird. Eine lange Wartezeit, nur um dann einen Programmdurchlauf zu machen, ist schlicht nervtötend.

3.4 Makefiles

1. Was sind Makefiles
2. Wie erstellt man Sie
   (a) von Hand
   (b) Automake

Kapitel 4

GUIs und C++

4.1 Was ist ein GUI?

GUI kommt vom englischen „Graphical User Interface“, was soviel heißt wie „Graphische Benutzerschnittstelle“. Grob gesagt ist es das, was der Benutzer von den meisten Programmen zu sehen bekommt. Also: Die Menüleiste, Textfelder, Buttons u.s.w.

Weiterhin gibt es natürlich noch das „Command Line Interface“ (kurz CLI, zu Deutsch "Kommandozeilenschnittstelle"), das vielen als Fenster mit schwarzen Hintergrund und weißer Schrift bekannt ist.

4.2 C++ und GUIs

C++ bringt von Haus aus kein GUI mit, da es als hardwarenahe plattformunabhängige Sprache erstellt wurde und GUIs stark vom verwendeten Betriebssystem abhängen. Dieser „Mangel“ könnte einige Neueinsteiger vielleicht auf den ersten Blick abschrecken, da sie sich einen Einstieg in C++ oder gar in die gesamte Programmierung mit vielen Buttons, Textfeldern, Statusanzeigen und Menüeintragen erhofft haben.

Wer jedoch einmal damit angefangen hat, kleinere Programme zu schreiben, wird schnell merken, dass der Reiz hier nicht von bunten Farben und blinkenden Symbolen ausgeht. Ganz im Gegenteil. Gerade für den Anfänger ist es am besten, wenn die eigentliche Funktion nicht durch Tonnen von "Design-Code" überdeckt

und unleserlich gemacht wird. Hat man erst einmal die Grundlagen erlernt, so ist es nicht schwer auch ansprechende Oberflächen zu erstellen.

Es gibt eine ganze Reihe von Bibliotheken, die einen weitgehend plattformübergreifende Programmierung mit C++ als Sprache ermöglichen. Aber auch die Programmierung über die API (für engl. „application programming interface“, deutsch: „Schnittstelle zur Anwendungsprogrammierung“) des spezifischen Betriebssystems ist natürlich möglich.

4.3 Einsatz von GUI

Sobald man mit C++ dann doch ein GUI programmieren will, muss auf externe Bibliotheken zurückgegriffen werden. Allen Arten von GUIs ist gemeinsam, dass sie ein Verständnis der Grundlagen von C++ erfordern. Nachfolgend sind einige Bibliotheken zur GUI-Programmierung aufgelistet. Viele der Bibliotheken lassen sich auch mit anderen Programmiersprachen einsetzen.

4.3.1 Qt

Qt ist eine leistungsstarke plattformübergreifende Klassenbibliothek, die von der norwegischen Firma Trolltech (inzwischen Qt Software und Teil von Nokia) entwickelt wird. Die Klassenbibliothek ist unter der GNU General Public License (GPL) und der GNU Lesser General Public License (LGPL) lizenziert. Es gibt außerdem eine proprietäre Lizenz, die allerdings lediglich zusätzlichen technischen Support beinhaltet.

Buchempfehlung
Das Buch „Qt für C++-Anfänger“ gibt einen Einstig in die Programmierung mit Qt. Es behandelt nicht viele Themen und ist somit wirklich nur zum Kennenlernen des Frameworks geeignet. Dafür sind die vorhandenen Seiten aber auch gut gefüllt.

4.3.2 GTK+

Das GIMP-Toolkit (abgekürzt: GTK+) ist eine freie Komponentenbibliothek unter der GNU Lesser General Public License (LGPL). Wie Qt ist es für viele Plattformen verfügbar.

Buchempfehlung
Das Buch „gtkmm für C++-Anfänger“ befindet sich noch im Anfangsstadium, es hat daher bislang nur wenige Inhalte.

4.3.3 wxWidgets

wxWidgets ist ein auf C++ basierendes Open-Source-Framework zur plattformunabhängigen Entwicklung von Anwendungen mit grafischer Benutzeroberfläche (GUI). Die wxWidgets-Lizenz ist eine leicht modifizierte LGPL und erlaubt daher ebenfalls die freie Verwendung in proprietärer und freier Software und den weiteren Vertrieb unter einer selbst gewählten Lizenz.

Kapitel 5

Hallo, du schöne Welt!

Es ist eine alte Tradition, eine neue Programmiersprache mit einem „Hello-World“-Programm einzuweihen. Auch dieses Buch soll mit der Tradition nicht brechen, hier ist das „Hello-World“-Programm in C++:

```
    1. include <iostream> // Ein- und Ausgabebibliothek


int main(){                                             // Hauptfunktion
    std::cout << "Hallo, du schöne Welt!" << std::endl; // Ausgabe
    return 0;                                           // Optionale Rückgabe an das
 Betriebssystem
}
```

Ausgabe

```
Hallo, du schöne Welt!
```

Zugegebenermaßen ist es nicht die Originalversion, sondern eine Originellversion von „Hello-World“. Wenn Sie das Programm ausführen, bekommen Sie den Text „Hallo, du schöne Welt!“ am Bildschirm ausgegeben. Sie wissen nicht, wie Sie das Programm ausführen können? Dann lesen Sie doch einmal das Kapitel über Compiler.

`#include <iostream>` stellt die nötigen Befehle zur Ein- und Ausgabe bereit. Als nächstes beginnt die „Hauptfunktion“ `main()`. Diese Hauptfunktion wird beim Ausführen des Programms aufgerufen. Sie ist also der zentrale Kern des Programms. Wenn Funktionen im Text erwähnt werden stehen dahinter übrigens immer Klammern, um sie besser von anderen Sprachbestandteilen wie beispielsweise Variablen unterscheiden zu können.

`std::cout` beschreibt den Standardausgabe-Strom. Dabei wird der Text meist in einem Terminal angezeigt (wenn die Ausgabe nicht in eine Datei oder an ein anderes Programm umgeleitet wird). Die beiden Pfeile (<<) signalisieren, dass der dahinterstehende Text auf die Standardausgabe „geschoben“ wird. Das `std::endl` gibt einen Zeilenumbruch aus und sorgt dafür das der Text jetzt am Bildschirm ausgegeben wird.

`return 0` beendet das Programm und zeigt dem Betriebssystem an, dass es erfolgreich ausgeführt wurde. Auf die Einzelheiten wird in den folgenden Kapiteln (oder Abschnitten) noch ausführlich eingegangen. Diese Zeile ist optional, wird Sie nicht angegeben, gibt der Compiler implizit 0 zurück.

Im Moment sollten Sie sich merken, dass jeder C++-Befehl mit einem Semikolon (`;`) abgeschlossen wird und dass geschweifte Klammern (`{...}`) Zusammengehörigkeit symbolisieren. So auch oben in der Hauptfunktion. Alles was zwischen den geschweiften Klammern steht, gehört zu ihr. Leerzeichen, Tabulatorzeichen und Zeilenumbrüche spielen für den C++-Compiler keine Rolle. Sie können das folgende Programm genauso übersetzen wie seine gut lesbare Version von weiter oben:

```
   1. include <iostream>


           int
  main    (
        )   {std        ::
    cout
<<         "Hallo, du schöne Welt!"<<std
  ::  endl

return 0;



    }
```

Ausgabe

```
Hallo, du schöne Welt!
```

Vorsichtig sollten Sie bei Zeilen sein die mit `#` beginnen. Leerzeichen und Tabulatoren sind zwar auch hier bedeutungslos, aber Zeilenumbrüche dürfen nicht stattfinden.

Es ist übrigens (für Ihren Rechner) auch irrelevant, ob Sie etwas wie `// Ein- und Ausgabebibliothek` mit in Ihr Programm schreiben oder nicht. Es handelt sich dabei um sogenannte Kommentare, die Sie in Kürze auch genauer kennenlernen werden. Beachten sollten Sie übrigens, dass bei C++ die Groß- und Kleinschreibung relevant ist. Schlüsselwörter und Namen der Standardbibliothek werden stets kleingeschrieben. Für die Groß-/Kleinschreibung von selbstdefinier-

ten Namen gibt es gewisse Konventionen, auf welche in einem späteren Kapitel eingegangen wird.

Hinweis

Bei allen in diesem Buch beschriebenen Programmen handelt sich um so genannte Kommandozeilenprogramme. Falls Sie eine IDE zur Entwicklung benutzten und das Programm direkt aus dieser heraus aufrufen, kann es Ihnen passieren, dass Sie nur einen kurzen Blitz von Ihrem Programm sehen, weil sich das Kommandozeilenfenster nach der Programmbeendigung sofort schließt.
In diesem Fall haben Sie 2 Optionen:

- Rufen Sie eine Kommandozeile auf und führen Sie das Programm von dort manuell aus. (empfohlen!)
- Schauen Sie nach ob es in Ihrer IDE eine Funktion gibt, die das Fenster nach Programmende noch einen Tastendruck lang offen hält. (Muss nicht vorhanden sein!)

Kapitel 6

Einfache Ein- und Ausgabe

Ein- und Ausgaberoutinen geben Ihnen die Möglichkeit, mit einem Programm zu interagieren. Dieses Kapitel beschäftigt sich mit der Eingabe über die Tastatur und der Ausgabe auf der Konsole in C++-typischer Form. Die C-Variante werden Sie später noch kennenlernen.

Um die C++ Ein- und Ausgabe nutzen zu können, müssen Sie die Bibliothek `iostream` einbinden. Das geschieht mit:

```
1. include <iostream>
```

Danach müssen die Befehle daraus bekannt gegeben werden, da sie sich in einem speziellen *Namensraum* befinden. Was Namensräume sind und wofür man sie einsetzt, werden Sie später noch erfahren. Um nun die Ein- und Ausgabebefehle nutzen zu können, müssen sie dem Compiler sagen: Benutze den Namensraum `std`. Da der Compiler das aber so nicht versteht, können Sie folgende gleichbedeutende Zeile benutzen:

```
using namespace std;
```

Der Namensraum „`std`“ heißt so viel wie Standard. Wenn etwas in diesem Namensraum steht, dann gehört es zur C++-Standardbibliothek und die sollten Sie (in der Regel) wann immer möglich einsetzen. In den Programmen dieses Buches wird der Namensraum immer direkt angegeben. Das heisst, wenn beispielsweise ein Objekt benutzen werden soll das im Namensraum „`std`“ liegt, wird ihm „`std::`“ vorangestellt. Die beiden Doppelpunkte heißen Bereichsoperator.

6.1 Einfache Ausgabe

Nun wollen wir aber endlich auch mal was Praktisches tun. Zugegebenermaßen nichts Weltbewegendes und im Grunde nicht einmal etwas wirklich Neues, denn Text haben wir ja schon im „Hello-World“-Programm ausgegeben.

```
    1. include <iostream>


int main(){
    std::cout << "Dieser Text steht nun in der Kommandozeile!";
    std::cout << "Dieser Text schließt sich direkt an...";
}
```

Ausgabe

```
Dieser Text steht nun in der Kommandozeile!Dieser Text schließt sich direkt an...
```

Wie der Text bereits selbst sagte, er erscheint in der Kommandozeile und der zweite schließt sich sehr direkt an. Wenn sie innerhalb einer Zeichenkette einen Zeilenumbruch einfügen möchten, gibt es zwei Möglichkeiten. Sie können die *Escape-Sequenz* `\n` in die Zeichenkette einfügen oder den *Manipulator* `endl` benutzen. Was genau Escape-Sequenzen oder Manipulatoren sind ist Thema eines späteren Kapitels, aber folgendes Beispiels demonstriert schon mal die Verwendung für den Zeilenumbruch:

```
    1. include <iostream>


int main(){
    std::cout << "Text in der Kommandozeile!\n";                 // Escape-Sequenz \n
    std::cout << "Dieser Text schließt sich an...\n";            // Das steht in einer eigenen
 Zeile
    std::cout << std::endl;                                      // Leerzeile mittels endl
    std::cout << "Text in der Kommandozeile!" << std::endl;      // Zeilenumbruch mit endl
    std::cout << "Dieser Text schließt sich an..." << std::endl; // Das steht in einer eigenen
 Zeile
}
```

Ausgabe

```
Text in der Kommandozeile!
Dieser Text schließt sich an...
Text in der Kommandozeile!
Dieser Text schließt sich an...
```

Beide Methoden haben scheinbar den gleichen Effekt, wo der kleine Unterschied liegt, ist allerdings auch Thema eines späteren Kapitels und im Moment noch nicht relevant.

6.2 Einfache Eingabe

Für die Eingabe muss ein wenig vorgegriffen werden, denn um etwas einzulesen, ist ja etwas nötig, worin das Eingelesene gespeichert werden kann. Dieser „Behälter“ nennt sich Variable. Eine Variable muss zunächst einmal angelegt werden, am besten lässt sich die Eingabe an einem Beispiel erklären:

```
    1. include <iostream>


int main(){
    int Ganzzahl;
    std::cout << "Benutzereingabe: ";
    std::cin >> Ganzzahl;
    std::cout << "Sie haben " << Ganzzahl << " eingegeben.";
}
```

Ausgabe

```
Benutzereingabe: 643
Sie haben 643 eingegeben.
```

Es wird, wie bereits erwähnt, erst eine Variable angelegt (`int Ganzzahl;`), welcher dann ein Wert zugewiesen wird (`cin >> Ganzzahl;`). Diese zweite Zeile bedeutet so viel wie: lies eine ganze Zahl von der Tastatur und speichere sie in der Variablen `Ganzzahl`.

`cin` ist sozusagen die Tastatur, `Ganzzahl` ist der Behälter und `>>` bedeutet so viel wie „nach“. Zusammen ergibt sich „Tastatur nach Behälter“, es wird also der „Inhalt“ der Tastatur in die Variable `Ganzzahl` verschoben. Dass eine Ganzzahl von der Tastatur gelesen wird, ist übrigens vom Datentyp der Variable abhängig, aber dazu später mehr.

Um den Inhalt der Variable wieder auszugeben, müssen Sie nichts weiter tun, als sie mit einem weiteren Schiebeoperator (`<<`) hinter `cout` anzuhängen. Es ist ohne Weiteres möglich, mehrere solcher Schiebeoperatoren hintereinander zu schalten, solange Sie nur die letzte Ein- oder Ausgabe mit einem Semikolon (`;`) abschließen. Bei der Eingabe muss natürlich der `>>`-Operator statt dem `<<`-Operator benutzt werden. Die Reihenfolge der Ein- oder Ausgabe bei solchen Konstruktionen

entspricht der eingegebenen Folge im Quelltext. Was zuerst hinter `cout` oder `cin` steht, wird also auch zuerst ausgeführt.

Fragen

Frage 1

Was ist hier verkehrt:
 	
`int Ganzzahl;`
`std::cin << Ganzzahl;`

Antwort
Der Operator muss >> sein.

Frage 2

Funktioniert folgendes Beispiel:
`1. include <iostream>`
 	
`int main(){`
`int Ganzzahl;`
`std::cin >> Ganzzahl;`
`std::cout << "Sie haben "`
`<< Ganzzahl`
`<< " eingegeben.";`
`}`

Antwort
Das Beispiel funktioniert hervorragend. Zeilenumbrüche sind kein Problem, solange sie nicht innerhalb einer Zeichenkette stehen. Wenn Sie allerdings beabsichtigen, einen Zeilenumbruch bei der Ausgabe zu erzeugen, müssen Sie `\n` oder `endl` benutzen.

Kapitel 7

Kommentare

In allen Programmiersprachen gibt es die Möglichkeit, im Quelltext Notizen zu machen. Für andere oder auch für sich selbst, denn nach ein paar Wochen werden Sie möglicherweise Ihren eigenen Quelltext nicht mehr ohne Weiteres verstehen. Kommentare helfen Ihnen und anderen besser und vor allem schneller zu verstehen, was der Quelltext bewirkt. In C++ gibt es zwei Varianten, um Kommentare zu schreiben:

```
// Ein Kommentar, der mit zwei Schrägstrichen eingeleitet wird, geht bis zum Zeilenende
/* Ein Kommentar dieser Art kann
   sich über mehrere Zeilen
   erstrecken */
```

Die erste, bis zum Zeilenende geltende Sorte, ist die moderne Art des Kommentars. Sie ist in der Regel vorzuziehen, da sie einige Vorteile gegenüber der alten, noch aus C stammenden Variante hat. Ein mehrzeiliger (manchmal auch als C-Kommentar bezeichneter) Kommentar sollte nur an Stellen verwendet werden, an denen längere Textpassagen stehen und möglichst nicht zwischen Code. Anwendung finden solche Kommentare oft am Dateianfang, um den Inhalt kurz zusammenzufassen oder Lizenzrechtliches zu regeln.

Der hauptsächliche Nachteil bei den mehrzeiligen Kommentaren besteht darin, dass man sie nicht „verschachteln“ kann. Oft werden beispielsweise Teile des Quellcodes zu Testzwecken kurzweilig auskommentiert. Folgendes Beispiel soll dies demonstrieren:

```
1. include <iostream> /* Ein- und Ausgabebibliothek */
```

```
int main(){                                             /* Hauptfunktion */
/*
    std::cout << "Hallo, du schöne Welt!" << std::endl; /* Ausgabe */

     • /


}
```

Das würde nicht funktionieren. Wenn hingegen die anderen Kommentare benutzt werden, gibt es solche Probleme nicht:

```
    1. include <iostream> // Ein- und Ausgabebibliothek


int main(){                                             // Hauptfunktion
    /*
    std::cout << "Hallo, du schöne Welt!" << std::endl; // Ausgabe
    */
}
```

Im ersten Beispiel wird die Einleitung von `/* Ausgabe */` einfach ignoriert. Der abschließende Teil beendet den Kommentar, der die Code-Zeile auskommentieren soll und der eigentliche Abschluss führt zu einem Kompilierfehler. Zugegeben, das ist nicht weiter schlimm, denn solch ein Fehler ist schnell gefunden, aber ihn von vorn herein zu vermeiden, ist eben noch zeitsparender. Im übrigen muss bei einzeiligen Kommentaren oft weniger geschrieben werden. Eine Ausnahme bilden Kommentare, die einfach zu lang sind, um sie auf eine Zeile zu schreiben. Dennoch sollten auch sie durch `//`-Kommentare realisiert werden.

```
viel Code...
// Ich bin ein Beispiel für einen langen Kommentar, der durch doppelte
// Schrägstriche über mehrere Zeilen geht. Würde ich am Dateianfang stehen,
// hätte man mich wahrscheinlich mit anderen Kommentarzeichen ausgestattet,
// aber da ich hier eindeutig von einer Riesenmenge Quelltext umgeben bin,
// hat man sich trotz der erhöhten Schreibarbeit für // entschieden
noch viel mehr Code...
```

Tip
Viele Texteditoren enthalten eine Tastenkombination, über die sich Text ein- und auskommentieren lässt. Besonders für längere Codepassagen ist dies nützlich.

Kapitel 8

Rechnen (lassen)

In diesem Kapitel soll unser Rechner einmal das tun, was er ohnehin am besten kann: Rechnen. Wir werden uns derweil zurücklehnen und zusehen oder besser gesagt, werden wir das tun, nachdem wir ihm mitgeteilt haben, was er rechnen soll.

8.1 Einfaches Rechnen

```
    1. include <iostream>


int main(){
    std::cout << "7 + 8 = " << 7 + 8 << std::endl; // Ausgabe einer Rechnung
}
```

Ausgabe

```
7 + 8 = 15
```

Zugegebenermaßen hätten Sie diese Rechnung wahrscheinlich auch im Kopf lösen können, aber warum sollten Sie sich so unnötig anstrengen. Ihr Rechner liefert doch auch das richtige Ergebnis und wenn Sie dies mit der Eingabe von Zahlen kombinieren, können Sie sogar bei jedem Programmdurchlauf zwei unterschiedliche Zahlen addieren:

```
    1. include <iostream>


int main(){
```

```
    int Summand1, Summand2;                    // Anlegen von 2 Variablen
    std::cin >> Summand1 >> Summand2;          // 2 Zahlen eingeben
    std::cout << Summand1 << " + " << Summand2 // beide durch " + " getrennt Wieder ausgeben
              << " = "                         // " = " ausgeben
              << Summand1 + Summand2           // Ergebnis berechnen und ausgeben
              << std::endl;                    // Zeilenumbruch
}
```

Ausgabe

```
Benutzereingabe: 774
Benutzereingabe: 123
774 + 123 = 897
```

Das Ergebnis lässt sich natürlich auch in einer Variable zwischenspeichern. Folgendes Beispiel demonstriert diese Möglichkeit:

```
    1. include <iostream>


int main(){
    int Summand1, Summand2, Ergebnis;          // Anlegen von 3 Variablen
    std::cin >> Summand1 >> Summand2;          // 2 Zahlen eingeben
    Ergebnis = Summand1 + Summand2;            // Ergebnis berechnen
    std::cout << Summand1 << " + " << Summand2 // beide durch " + " getrennt wieder ausgeben
              << " = "                         // " = " ausgeben
              << Ergebnis                      // Ergebnis ausgeben
              << std::endl;                    // Zeilenumbruch
}
```

Ausgabe

```
Benutzereingabe: 400
Benutzereingabe: 300
400 + 300 = 700
```

8.2 Die großen 4

C++ beherrscht die 4 Grundrechenarten: Addition (+), Subtraktion (-), Multiplikation (*) und Division (/). Genau wie in der Mathematik gilt auch in C++ die Regel Punktrechnung geht vor Strichrechnung und Klammern gehen über alles. Das folgende Beispiel soll eine komplexere Rechnung demonstrieren:

```
    1. include <iostream>
```

```
int main(){
    int Ergebnis;                          // Anlegen einer Variable
    Ergebnis = ((3+3*4)/5-1)*512-768;      // Ergebnis berechnen
    std::cout << "((3+3*4)/5-1)*512-768 = " // Aufgabe ausgeben
              << Ergebnis                  // Ergebnis ausgeben
              << std::endl;                // Zeilenumbruch
}
```

Ausgabe

```
((3+3*4)/5-1)*512-768 = 256
```

Gerechnet wird in dieser Reihenfolge:

```
3 *   4 =   12
    3 +  12 =   15
   15 /   5 =    3
    3 -   1 =    2
    2 * 512 = 1024
 1024 - 768 =  256
```

Sie sollten darauf achten, immer die gleiche Anzahl öffnende und schließende Klammern zu haben, denn dies ist ein beliebter Fehler, der von Anfängern meist nicht so schnell gefunden wird. Compiler bringen in solchen Fällen nicht selten Meldungen, die einige Zeilen unter dem eigentlichen Fehler liegen.

8.3 Zusammengesetzte Operatoren

C++ ist eine Sprache für schreibfaule Menschen, daher gibt es die Möglichkeit, die Rechenoperatoren mit dem Zuweisungsoperator zu kombinieren. Dies sieht dann folgendermaßen aus:

```
Zahl  = 22;
Zahl += 5; // Zahl = Zahl + 5;
Zahl -= 7; // Zahl = Zahl - 7;
Zahl *= 2; // Zahl = Zahl * 2;
Zahl /= 4; // Zahl = Zahl / 4;
```

Als Kommentar sehen Sie die Langfassung geschrieben. Diese Kurzschreibweise bedeutet nicht mehr, als dass die vor (!) dem Zuweisungsoperator stehende Rechenoperation mit der Variable auf der linken Seite und dem Wert auf der rechten

Seite ausgeführt und das Ergebnis der Variable auf der linken Seite zugewiesen wird. Sie sollten diese Kurzschreibweise der ausführlichen vorziehen, da sie nicht nur die Finger schont, sondern auch noch ein wenig schneller ist.

Am besten werden Sie dies wahrscheinlich verstehen, wenn Sie es einfach ausprobieren. Stehen auf der rechten Seite noch weitere Rechenoperationen, so werden diese zuerst ausgeführt. Das Ganze stellt sich dann also folgendermaßen dar:

```
Zahl  = 5;
Zahl *= 3+4; // Zahl = Zahl * (3+4);
```

8.4 Inkrement und Dekrement

Inkrementieren bedeutet, den Wert einer Variable um 1 zu erhöhen, entsprechend bedeutet dekrementieren 1 herunterzählen. Dem Inkrementoperator schuldet C++ übrigens seinen Namen. Die beiden Operatoren gibt es jeweils in der Präfix und der Postfix Variante. Insgesamt ergeben sich also 4 Operatoren:

```
Zahl = 5;
Zahl++; // Inkrement Postfix (Zahl == 6)
++Zahl; // Inkrement Präfix  (Zahl == 7)
Zahl-; // Dekrement Postfix (Zahl == 6)
-Zahl; // Dekrement Präfix  (Zahl == 5)
```

Der Unterschied zwischen Inkrement (++) und Dekrement (−) ist sofort ohne größeres Nachdenken erkennbar. Der Sinn von Präfix und Postfix ergibt sich hingegen nicht sofort von selbst. C++ schuldet seinen Namen der Postfix-Variante.

Der Unterschied zwischen Präfix und Postfix besteht im Rückgabewert. Die Präfix-Variante erhöht den Wert einer Zahl um 1 und gibt diesen neuen Wert zurück. Die Postfix-Variante erhöht den Wert der Variable ebenfalls um 1, gibt jedoch den Wert zurück, den die Variable vor der Erhöhung hatte.

Das folgende kleine Programm zeigt den Unterschied:

```
1. include <iostream>


int main(){
```

```
    int Zahl;                           // Anlegen einer Variable
    std::cout << "Zahl direkt ausgeben:\n";
    Zahl = 5;                           // Zahl den Wert 5 zuweisen
    std::cout << Zahl << ' ';           // Zahl ausgeben
    Zahl++;                             // Inkrement Postfix (Zahl == 6)
    std::cout << Zahl << ' ';           // Zahl ausgeben
    ++Zahl;                             // Inkrement Präfix  (Zahl == 7)
    std::cout << Zahl << ' ';           // Zahl ausgeben
    Zahl-;                             // Dekrement Postfix (Zahl == 6)
    std::cout << Zahl << ' ';           // Zahl ausgeben
    -Zahl;                             // Dekrement Präfix  (Zahl == 5)
    std::cout << Zahl << ' ';           // Zahl ausgeben
    std::cout << "\nRückgabewert des Operators ausgeben:\n";
    Zahl = 5;                           // Zahl den Wert 5 zuweisen
    std::cout << Zahl   << ' ';         // Zahl ausgeben
    std::cout << Zahl++ << ' ';         // Inkrement Postfix (Zahl == 6)
    std::cout << ++Zahl << ' ';         // Inkrement Präfix  (Zahl == 7)
    std::cout << Zahl- << ' ';         // Dekrement Postfix (Zahl == 6)
    std::cout << -Zahl << ' ';         // Dekrement Präfix  (Zahl == 5)
    std::cout << "\nEndwert von Zahl: " << Zahl << std::endl;
}
```

Ausgabe

```
Zahl direkt ausgeben:
5 6 7 6 5
Rückgabewert des Operators ausgeben:
5 5 7 7 5
Endwert von Zahl: 5
```

Thema wird Später näher erläutert...
In einem späteren Kapitel werden Sie noch ein paar zusätzliche Informationen erhalten, was beim Rechnen schief gehen kann und wie Sie es vermeiden. Wenn Sie beim Herumexperimentieren mit Rechenoperationen plötzlich scheinbar unerklärliche Ergebnisse erhalten, dann ist es an der Zeit eine Blick auf dieses Kapitel zu werfen.

Aufgaben

Aufgabe 1

Rechnen Sie die Ergebnisse die folgenden Aufgaben aus und schreiben Sie ein Programm, welches das Gleiche tut.

Zahl = (500-100*(2+1))*5 Zahl = (Zahl-700)/3 Zahl += 50*2 Zahl *= 10-8 Zahl /= Zahl-200

Musterlösung

```
    1. include <iostream>



int main(){
    int Zahl;


    Zahl  = (500-100*(2+1))*5;  //1000
    Zahl  = (Zahl-700)/3;       //100
    Zahl += 50*2;               //200
    Zahl *= 10-8;               //400
    Zahl /= Zahl-200;           //2


    std::cout << "Zahl: " << Zahl;
}
```

Ausgabe

```
Zahl: 2
```

Kapitel 9

Variablen, Konstanten und ihre Datentypen

Variablen sind Behälter für Werte, sie stellen gewissermaßen das Gedächtnis eines Programms bereit. Konstanten sind Variablen, die ihren Wert nie verändern. Der Datentyp einer Variable oder Konstante beschreibt, wie der Inhalt zu verstehen ist. Ein Rechner kennt nur zwei Zustände: 0 und 1. Durch eine Aneinanderreihung solcher Zustände lassen sich verschiedene Werte darstellen. Mit 8 aufgereihten Zuständen (= 8 Bit = 1 Byte) lassen sich bereits 256 verschiedene Werte darstellen. Diese Werte kann man als Ganzzahl, Zeichen, Wahrheitswert oder Gleitkommazahl interpretieren. Der Datentyp gibt Auskunft darüber, um was es sich handelt.

9.1 Datentypen

Zunächst sollen die Datentypen von C++ beschrieben werden, denn sie sind grundlegend für eine Variable oder Konstante. Es gibt 4 Gruppen von Datentypen: Wahrheitswerte, Zeichen, Ganzzahlen und Gleitkommazahlen.

9.1.1 Wahrheitswerte

Der Datentyp für Wahrheitswerte heißt in C++ `bool`, was eine Abkürzung für Boolean ist. Er kann nur 2 Zustände annehmen: `true` (Wahr) oder `false` (falsch). Obwohl eigentlich 1 Bit ausreichen würde, hat `bool` mindestens eine Größe von einem Byte (also 8 Bit), denn 1 Byte ist die kleinste adressierbare Einheit und

somit die Minimalgröße für jeden Datentyp. Es ist auch durchaus möglich, dass ein `bool` beispielsweise 4 Byte belegt, da dies auf einigen Prozessorarchitekturen die Zugriffsgeschwindigkeit erhöht.

9.1.2 Zeichen

Zeichen sind eigentlich Ganzzahlen. Sie unterscheiden sich von diesen nur bezüglich der Ein- und Ausgabe. Jeder Zahl ist ein Zeichen zugeordnet, mit den Zahlen lässt sich ganz normal rechnen, aber bei der Ausgabe erscheint das zugeordnete Zeichen auf dem Bildschirm. Welches Zeichen welcher Zahl zugeordnet ist, wird durch den verwendeten Zeichensatz festgelegt.

Die meisten Zeichensätze beinhalten den sogenannten *ASCII-Code* (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), welcher die Zeichen 0 – 127 belegt. Er enthält 32 Steuerzeichen (0 – 31) und 96 druckbare Zeichen (32 – 127).

`char` ist der Standard-Datentyp für Zeichen. Er ist in der Regel 1 Byte groß und kann somit 256 verschieden Zeichen darstellen. Diese genügen für einen erweiterten ASCII-Code, welcher zum Beispiel auch deutsche Umlaute definiert. `wchar_t` ist ein Datentyp für Unicodezeichen und hat gewöhnlich eine Größe von 2 Byte, ist aber zunehmend auch mit 4 Byte zu finden. Die folgende Liste enthält einige nützliche Links zu Artikeln der Wikipedia:

- ASCII
- ISO 8859-1 (Latin-1)
- Codepage 437 (DOS / DOS-Box)
- Unicode

9.1.3 Ganzzahlen

Für ganze Zahlen sind die Datentypen `short`, `int` und `long` definiert. Weiterhin sind, wie schon gesagt, auch `char` und `wchar_t` ganzzahlige Datentypen. Mit Ausnahme von `wchar_t` kann jedem dieser Datentypen ein `signed` oder `unsigned` vorangestellt werden. `signed` bedeutet mit, `unsigned` ohne Vorzeichen, entsprechend hat der Datentyp dann einen negativen und einen positiven (`signed`) oder nur einen positiven (`unsigned`) Wertebereich, welcher dann aber doppelt so groß ist. `wchar_t` entspricht meistens dem Datentyp `unsigned short`, dies ist in ISO-C++ aber nicht vorgeschrieben.

ISO-C++ schreibt auch die genaue Größe der Datentypen nicht vor, es gibt lediglich die Reihenfolge bezüglich der Größe vor: `char <= short <= int <= long`

Außerdem ist festgelegt, dass `short` mindestens 2 Byte und `long` mindestens 4 Byte lang sein müssen. `int` hat (üblicherweise) auf 16-Bit-Rechnern eine Größe von 2 Byte und auf 32-Bit-Rechnern eine Größe von 4 Byte. Die nachfolgende Tabelle zeigt die ganzzahligen Datentypen mit ihrer *üblichen Größe* und dem entsprechendem Wertebereich:

Typ	Speicherplatz	Wertebereich (dezimal)
`char`	1 Byte	-128 bis +127 bzw. 0 bis 255
`signed char`	1 Byte	-128 bis +127
`unsigned char`	1 Byte	0 bis 255
`short`	2 Byte	-32768 bis +32767
`unsigned short`	2 Byte	0 bis 65535
`int`	4 Byte	-2147483648 bis +2147483647
`unsigned int`	4 Byte	0 bis 4294967295
`long`	4 Byte	-2147483648 bis +2147483647
`unsigned long`	4 Byte	0 bis 4294967295

Auffällig ist, dass in der Tabelle nur für `char` eine `signed` Variante vorkommt. Das liegt daran, dass die übrigen Datentypen ohne den Zusatz `signed` immer vorzeichenbehaftet sind, für `char` ist dies hingegen nicht festgelegt. Dennoch können Sie das Schlüsselwort `signed` natürlich auch in Verbindung mit den anderen Datentypen (`signed short`, `signed int` und `signed long`) benutzen. In bestimmten Situationen kann dies zum Beispiel helfen die Übersicht zu erhöhen. Wenn Sie `char` für Zeichen benutzen, ist eine Angabe von `signed` oder `unsigned` nicht nötig, möchten Sie eine Variable dieses Datentyps zum Rechnen benutzen, sollten Sie die Schlüsselworte hingegen angeben.

Das waren jetzt viele Informationen auf wenig Raum. Merken Sie sich einfach in etwa die Größe der 4 Datentypen, den Wertebereich können Sie dann entsprechend ableiten ($2^{\text{Anzahl der Bits}}$, 1 Byte = 8 Bit). Merken Sie sich weiterhin das `signed` vorzeichenbehaftet, `unsigned` vorzeichenlos bedeutet und dass im Falle einer fehlenden Angabe `signed` angenommen wird.

Wählen Sie ein `int`, wenn dieser Typ alle Zahlen des nötigen Wertebereichs aufnehmen kann, bei vorzeichenlosen Zahlen verwenden Sie `unsigned`. Reicht dieser Wertebereich nicht aus und ist `long` größer, dann nehmen Sie `long` (bzw.

`unsigned long`). `short` und `unsigned short` sollte nur Verwendung finden, wenn Speicherplatz knapp ist, etwa bei Verwendung großer Arrays, oder wenn low-level-Datenstrukturen festgelegter Größe benutzt werden müssen. Achten Sie darauf, dass der theoretisch größte Wert, welcher in Ihrer Variable gespeichert wird, den größten möglichen Wert nicht überschreitet. Selbiges gilt natürlich auch für die Unterschreitung des kleinstmöglichen Wertes.

Ein Unter- oder Überlauf ist übrigens durchaus möglich. Die meisten Compiler bieten zwar eine Option an, um in einem solchem Fall einen Fehler zu erzeugen, aber diese Option ist standardmäßig nicht aktiv. Im folgenden kleinen Beispiel werden die Datentypen `short` (min: -32768, max: 32767) und `unsigned short` benutzt und bei beiden wird je ein Unter- und ein Überlauf ausgeführt:

```
    1. include <iostream>


int main(){
    short Variable1=15000;
    unsigned short Variable2=15000;
    std::cout << "short Variable:          " << Variable1 << std::endl
              << "unsigned short Variable: " << Variable2 << std::endl
              << "+30000\n\n";
    Variable1 += 30000;
    Variable2 += 30000;
    std::cout << "short Variable:          " << Variable1 << " (Überlauf)" << std::endl
              << "unsigned short Variable: " << Variable2 << std::endl
              << "+30000\n\n";
    Variable1 += 30000;
    Variable2 += 30000;
    std::cout << "short Variable:          " << Variable1 << std::endl
              << "unsigned short Variable: " << Variable2 << " (Überlauf)" << std::endl
              << "-30000\n\n";
    Variable1 -= 30000;
    Variable2 -= 30000;
    std::cout << "short Variable:          " << Variable1 << std::endl
              << "unsigned short Variable: " << Variable2 << " (Unterlauf)" << std::endl
              << "-30000\n\n";
    Variable1 -= 30000;
    Variable2 -= 30000;
    std::cout << "short Variable:          " << Variable1 << " (Unterlauf)" << std::endl
              << "unsigned short Variable: " << Variable2 << std::endl;
}
```

Ausgabe

```
short Variable:          15000
unsigned short Variable: 15000
+30000
short Variable:          -20536 (Überlauf)
unsigned short Variable: 45000
+30000
short Variable:          9464
unsigned short Variable: 9464 (Überlauf)
-30000
short Variable:          -20536
unsigned short Variable: 45000 (Unterlauf)
-30000
short Variable:          15000 (Unterlauf)
unsigned short Variable: 15000
```

Verständlicher wird dieses Phänomen, wenn man die Zahlen binär (Duales Zahlensystem) darstellt. Der Einfachheit halber beginnen wir mit der Darstellung der `unsigned short` Variable:

```
Addition im Dualsystem mit {{cpp|unsigned short}} als Datentyp

Rechnung 1
        |0011101010011000| 15000
     +  |0111010100110000| 30000
---------------------
Merker  |111      11     |
---------------------
     =  |1010111111001000| 45000

Rechnung 2
        |1010111111001000| 45000
     +  |0111010100110000| 30000
---------------------
Merker 1|1111111         |
---------------------
     = 1|0010010011111000| 9464
```

Die beiden Rechnungen weisen keinerlei Besonderheiten auf. Da nur die letzten 16 Ziffern beachtet werden (2 Byte = 16 Bit), entfällt in der zweiten Rechnung die 1 vor dem Horizontalstrich, wodurch das Ergebnis (in dezimaler Schreibweise) 9464 und nicht 75000 lautet.

```
Subtraktion im Dualsystem mit {{cpp|unsigned short}} als Datentyp

Rechnung 3
          |0010010011111000| 9464
     -    |0111010100110000| 30000
-----------------------
Merker...1|1111111         |
-----------------------
     =...1|1010111111001000| 45000

Rechnung 4
          |1010111111001000| 45000
     -    |0111010100110000| 30000
-----------------------
Merker    |111      11     |
-----------------------
     =    |0011101010011000| 15000
```

In diesem Fall ist die zweite Rechnung unauffällig. In der ersten Rechnung wird hingegen eine große Zahl von einer kleineren angezogen, was zu einem negativen Ergebnis führt oder besser führen würden, denn die Untergrenze ist in diesem Fall ja 0. Dort wo die 3 Punkte stehen, folgt eine unendliche Anzahl von Einsen. Dies ist keineswegs nur bei Dualzahlen der Fall, wenn Sie im dezimalen System eine große Zahl von einer kleineren nach den üblichen Regeln der schriftlichen Subtraktion abziehen, so erhalten Sie ein ähnliches Ergebnis:

```
24
     -    31
--------
Merker...1
--------
     =...993
```

Die duale Darstellung der 4 Rechnungen mit der `short`-Variable wird Ihnen sehr bekannt vorkommen:

```
Addition im Dualsystem mit {{cpp|short}} als Datentyp

Rechnung 1
        |0|0111010100110000| 15000
     +  |0|1110101001100000| 30000
----------------------
Merker  |1|11      11     |
----------------------
     =  |1|0101111111001000| -20536
```

```
Rechnung 2
        |1|0101111111001000| -20536
     +  |0|1110101001100000| 30000
----------------------
Merker 1|1|111111         |
----------------------
     = 1|0|0100100111111000| 9464

Subtraktion im Dualsystem mit {{cpp|short}} als Datentyp

Rechnung 3
          |0|0100100111111000| 9464
     -    |0|1110101001100000| 30000
------------------------
Merker...1|1|111111         |
------------------------
     =...1|1|0101111111001000| -20536

Rechnung 4
          |1|0101111111001000| -20536
     -    |0|1110101001100000| 30000
------------------------
Merker    |1|11      11     |
------------------------
     =    |0|0111010100011000| 15000
```

Die dualen Ziffern sind die gesamte Zeit über exakt die gleichen, der einzige Unterschied besteht darin, dass die erste Ziffer (welche durch einen weiteren Horizontalstrich abgegrenzt wurde) nun als Vorzeichenbit interpretiert wird (`0` – Positiv, `1` – Negativ). Dies führt zu einer veränderten Darstellung im Dezimalsystem.

9.1.4 Gleitkommazahlen

Eine Gleitkommavariable kann sich eine bestimmte Anzahl Ziffern merken und dazu die Position des Kommas. Das Wissen über den internen Aufbau einer solchen Zahl werden Sie wahrscheinlicher eher selten bis nie brauchen, daher sei an dieser Stelle auf den Wikipediaartikel über Gleitkommazahlen verwiesen. In C++ werden Sie Gleitkommazahlen/-variable für das Rechnen mit Kommazahlen verwenden. Es gibt 3 Datentypen für Gleitkommazahlen, die in der folgenden Tabelle mit ihren üblichen Werten aufgelistet sind:

Typ	Speicherplatz	Wertebereich	kleinste Positive Zahl	Genauigkeit
`float`	4 Byte	$\pm 3,4 \cdot 10^{38}$	$1,2 \cdot 10^{-38}$	6 Stellen
`double`	8 Byte	$\pm 1,7 \cdot 10^{308}$	$2,3 \cdot 10^{-308}$	15 Stellen
`long double`	10 Byte	$\pm 1,1 \cdot 10^{4932}$	$3,4 \cdot 10^{-4932}$	19 Stellen

Die Auswahl eines Gleitkommadatentyps ist weniger einfach als die einer Ganzzahl. Wenn Sie nicht genau wissen, was Sie nehmen sollen, ist `double` in der Regel eine gute Wahl. Sobald Sie erst einmal ausreichend Erfahrung haben, wird es Ihnen leichter fallen abzuschätzen, ob `float` oder `long double` für Ihr Problem vielleicht eine bessere Wahl sind.

9.2 Variablen

Bevor eine Variable verwendet werden kann, muss sie dem Compiler bekannt gegeben werden. Dies bezeichnet man als Deklaration der Variable.

Im vorherigen Kapitel haben wir bereits mit Variablen vom Typ `int` gerechnet. Nun sollen Sie lernen, wie Variablen in C++ angelegt werden. Die allgemeine Syntax lautet: `Datentyp Name;`

Außerdem ist es möglich, mehrere Variablen des gleichen Typs hintereinander anzulegen: `Datentyp Variable1, Variable2, Variable3;`

Auch kann man einer Variable einen Anfangswert geben, dies bezeichnet man als Initialisierung. Es gibt 2 syntaktische Möglichkeiten (Schreibweisen) für Initialisierungen, welche anhand einer `int` Variable gezeigt werden soll:

```
int Zahl=100;  // Möglichkeit 1
int Zahl(100); // Möglichkeit 2
```

Die erste Variante ist weit verbreitet, aber nicht besser. Bei den fundamentalen Datentypen von C++ spielt es keine Rolle welche Variante Sie verwenden, aber bei komplexeren Datentypen (Klassen) kann es zu Verwechslungen mit dem Zuweisungsoperator kommen, wenn Sie Möglichkeit 1 benutzen. Den genauen Unterschied zwischen einer Initialisierung und einer Zuweisung werden Sie kennen lernen, sobald es um Klassen geht. Für den Moment sollten Sie sich für eine Variante entscheiden.

Für Möglichkeit 1 spricht die große Verbreitung und die damit verbundene „gewohnte Benutzung“. Für Möglichkeit 2 spricht hingegen die bessere Lesbarkeit sobald man sich daran gewöhnt hat.

Variablen mit Anfangswerten können natürlich auch hintereinander angelegt werden, sofern Sie den gleichen Datentyp besitzen:

```
int Zahl1(77), Zahl2, Zahl3=58; // 3 int-Variablen von denen 2 Anfangswerte haben
```

Wenn Sie einer Variable keinen Anfangswert geben, müssen Sie ihr später im Programm noch einen Wert zuweisen, bevor Sie mit ihr arbeiten (also damit rechnen oder den Inhalt ausgeben lassen). Weisen Sie einer solchen Variable keinen Wert zu und benutzen sie, so ist der Inhalt zufällig. Genaugenommen handelt es sich dann um die Bitfolge, die an der Stelle im Speicher stand, an der Ihre Variable angelegt wurde.

Das nachfolgende kleine Programm zeigt dies:

```
    1. include <iostream> // Ein-/Ausgabe


int main(){
    int Zahl;                                           // Ganzzahlige Variable
    double Kommazahl1, Kommazahl2;                      // Gleitkommavariablen
    char Zeichen;                                       // Zeichenvariable
    std::cout << "Zahl: "       << Zahl       << std::endl  // Ausgabe der Werte
              << "Kommazahl1: " << Kommazahl1 << std::endl  // welche jedoch
              << "Kommazahl2: " << Kommazahl2 << std::endl  // nicht festgelegt
              << "Zeichen: "    << Zeichen    << std::endl; // wurden
}
```

Ausgabe

```
Zahl: -1211024315
Kommazahl1: 4.85875e-270
Kommazahl2: -3.32394e-39
Zeichen: f
```

Die Ausgabe lautet bei jedem Ausführen des Programms anders. Sollte dies bei Ihnen nicht der Fall sein, so stehen nur zufällig die gleichen Werte an der Stelle im Speicher welchen die jeweilige Variable belegt. Variablen keinen Anfangswert zu geben ist beispielsweise sinnvoll, wenn Sie vorhaben über `cin` einen Wert in die Variable einzulesen.

```
    1. include <iostream> // Ein-/Ausgabe


int main(){
    int Zahl;                                           // Ganzzahlige Variable
    double Kommazahl1, Kommazahl2;                      // Gleitkommavariablen
    char Zeichen;                                       // Zeichenvariable
    std::cout << "Geben Sie bitte durch Leerzeichen getrennt eine Ganzzahl, 2 Kommazahlen "
            "und ein Zeichen ein:\n";
    std::cin >> Zahl                                    // Eingabe von Werten
             >> Kommazahl1                              // mit denen die 4
             >> Kommazahl2                              // Variablen gefüllt
             >> Zeichen;                                // werden
    std::cout << "Zahl: "       << Zahl       << std::endl  // Ausgabe der Werte
              << "Kommazahl1: " << Kommazahl1 << std::endl  // welche zuvor
              << "Kommazahl2: " << Kommazahl2 << std::endl  // eingegeben
              << "Zeichen: "    << Zeichen    << std::endl; // wurden
}
```

Ausgabe

```
Geben Sie bitte durch Leerzeichen getrennt eine Ganzzahl, 2 Kommazahlen und ein Zeichen ein:
Benutzereingabe: 6 8.4 6.0 g
Zahl: 6
Kommazahl1: 8.4
Kommazahl2: 6
Zeichen: g
```

9.3 Konstanten

Konstanten sind, wie schon oben beschrieben, Variablen, welche ihren Wert nicht verändern. Daraus folgt logisch, dass eine Konstante immer mit einem Anfangswert initialisiert werden muss, andernfalls hätten Sie eine Konstante mit einem zufälligen Wert und das ergibt keinen Sinn. Das Schlüsselwort, um eine Variable zu einer Konstante zu machen, ist `const`. Es gehört immer zu dem, was links davon steht, es sei denn, links steht nichts mehr, dann gehört es zu dem Teil auf der rechten Seite. Dies klingt zwar kompliziert, ist es aber eigentlich gar nicht. Für uns bedeutet es im Moment nur, dass Sie 2 Möglichkeiten haben, eine Variable zu einer Konstante zu machen:

```
const int Zahl(400); // Alternativ: const int Zahl=400;
// oder
int const Zahl(400); // Alternativ: int const Zahl=400;
```

Beides hat die gleiche Wirkung, wieder ist die erste Variante weit verbreitet und wieder ist die zweite Variante der besseren Lesbarkeit bei komplexeren Datentypen (Arrays von Zeigern Konstante auf Memberfunktionen...) vorzuziehen. Entscheiden Sie sich für die Variante, die Ihnen besser gefällt und verwenden Sie diese. Wichtig ist, dass Sie der Variante, für die Sie sich entscheiden, treu bleiben, wenigstens für die Dauer eines Projekts. Denn Code, in dem sich der Schreibstil ständig ändert, ist schwerer zu lesen, als alles andere.

9.4 Literale und ihre Datentypen

Ein Literal ist eine Zeichenkette, die zu Darstellung des Wertes einer der oben beschriebenen Datentypen dient.

```
1. include <iostream> // Ein-/Ausgabe


int main() {
    std::cout << 100  << std::endl  // 100 ist ein int-Literal
              << 4.7  << std::endl  // 4.7 ist ein double-Literal
              << 'h'  << std::endl  // 'h' ist ein char-Literal
              << true << std::endl; // true ist ein bool-Literal
}
```

Ausgabe

```
100
4.7
h
1
```

Bei der Ausgabe ist zu beachten, dass Boolean-Werte als 0 (*false*) bzw. 1 (*true*) ausgeben werden.

Da die Bestimmung des Typs für einen ganzzahligen Wert etwas schwieriger ist als bei den übrigen, werden wir diese zuletzt behandeln. Bei Gleitkommaliteralen ist festgelegt, dass es sich um `double`-Werte handelt. Um einen Gleitkommaliteral mit einem anderen Typ zu erhalten, ist ein so genanntes Suffix nötig.

```
5.0  // double
6.7f // float
2.6F // float
9.4l // long double
4.0L // long double
```

Eine Zahl mit Komma (`.`) ist also ein `double`-Wert. Folgt der Zahl ein `f` oder ein `F` wird sie zu einem `float`-Wert und folgt ihr ein `l` oder ein `L` wird sie zu einem `long double`-Wert. Gleitpunktzahlen können auch in der wissenschaftlichen Schreibweise dargestellt werden.

```
5.7e10
3.3e-3
8.7e666L
4.2e-4F
```

Wie die letzten beiden Beispiele zeigen, können auch hierbei die Suffixe für den Dateityp genutzt werden. Um ein Zeichen beziehungsweise eine Zeichenketten als `wchar_t` zu kennzeichnen, stellt man ihr ein `L` voran:

```
'a'                 // char
L'b'                // wchar_t
"Ich bin ein Text"  // char*
L"Ich bin ein Text" // wchar_t*
```

Was das Sternchen (`*`) hinter dem Datentyp im Kommentar bedeutet, werden Sie in einem späteren Kapitel erfahren. `bool` kann nur 2 Zustände annehmen, entsprechend gibt es auch nur 2 `bool`-Literale: `true` und `false`.

Nun zu den Ganzzahlen. Neben der dezimalen Darstellung von Zahlen gibt es in C++ auch die Möglichkeit der Oktalen und Hexadezimalen Darstellung. Um eine Zahl als Oktal zu kennzeichnen, wird ihr eine `0` (Null) vorangestellt, für eine Hexadezimalzahl wird `0x` (zu empfehlen, da deutlich besser Lesbar) oder `0X` vorangestellt. Die Groß-/Kleinschreibung der hexadezimalen Ziffern spielt keine Rolle.

```
756        // Dezimal,     Dezimal: 756
046        // Oktal,       Dezimal: 38
```

```
0757       // Oktal,       Dezimal: 495
0xffff     // Hexadezimal, Dezimal: 65535
0x1234ABcd // Hexadezimal, Dezimal: 305441741
```

Der Datentyp wird durch die Größe des Wertes bestimmt, wobei die folgende Reihenfolge gilt: `int`, `unsigned int`, `long`, `unsigned long`. Weiterhin kann jeder Ganzzahl das Suffix `u` oder `U` für `unsigned` und `l` oder `L` für `long` angehängt werden. Auch beide Suffixe gleichzeitig sind möglich. Die Reihenfolge ändert sich entsprechend den durch die Suffixe festgelegten Kriterien.

Das Wissen über die Datentypen von Literalen werden Sie wahrscheinlich eher selten benötigen, daher reicht es „mal etwas davon gehört zu haben“ und es, wenn nötig, nachzuschlagen.

Kapitel 10

Rechnen mit unterschiedlichen Datentypen

Sie kennen nun die Datentypen in C++ und haben auch schon mit `int`-Variablen gerechnet. In diesem Kapitel erfahren Sie, wie man mit Variablen unterschiedlichen Typs rechnet. Es geht also weniger um das Ergebnis selbst, als viel mehr darum wie der Ergebnisdatentyp lautet.

10.1 Ganzzahlen unter sich

Das Rechnen mit Ganzzahlen ist leicht zu begreifen. Die „kleinen“ Datentypen werden als `int` behandelt. Bei den größeren entscheidet der größte Datentyp über den Ergebnistyp. Die folgende Liste zeigt die Zusammenhänge:

```
char          + char          => int           | wchar_t       + char          => int
char          + wchar_t       => int           | wchar_t       + wchar_t       => int
char          + signed char   => int           | wchar_t       + signed char   => int
char          + unsigned char => int           | wchar_t       + unsigned char => int
char          + short         => int           | wchar_t       + short         => int
char          + unsigned short => int          | wchar_t       + unsigned short => int
char          + int           => int           | wchar_t       + int           => int
char          + unsigned int  => unsigned int  | wchar_t       + unsigned int  => unsigned int
char          + long          => long          | wchar_t       + long          => long
char          + unsigned long => unsigned long | wchar_t       + unsigned long => unsigned long

signed char   + char          => int           | unsigned char + char          => int
signed char   + wchar_t       => int           | unsigned char + wchar_t       => int
signed char   + signed char   => int           | unsigned char + signed char   => int
signed char   + unsigned char => int           | unsigned char + unsigned char => int
signed char   + short         => int           | unsigned char + short         => int
signed char   + unsigned short => int          | unsigned char + unsigned short => int
signed char   + int           => int           | unsigned char + int           => int
```

```
signed char    + unsigned int   => unsigned int  | unsigned char  + unsigned int   => unsigned int
signed char    + long           => long          | unsigned char  + long           => long
signed char    + unsigned long  => unsigned long | unsigned char  + unsigned long  => unsigned long

short          + char           => int           | unsigned short + char           => int
short          + wchar_t        => int           | unsigned short + wchar_t        => int
short          + signed char    => int           | unsigned short + signed char    => int
short          + unsigned char  => int           | unsigned short + unsigned char  => int
short          + short          => int           | unsigned short + short          => int
short          + unsigned short => int           | unsigned short + unsigned short => int
short          + int            => int           | unsigned short + int            => int
short          + unsigned int   => unsigned int  | unsigned short + unsigned int   => unsigned int
short          + long           => long          | unsigned short + long           => long
short          + unsigned long  => unsigned long | unsigned short + unsigned long  => unsigned long

int            + char           => int           | unsigned int   + char           => unsigned int
int            + wchar_t        => int           | unsigned int   + wchar_t        => unsigned int
int            + signed char    => int           | unsigned int   + signed char    => unsigned int
int            + unsigned char  => int           | unsigned int   + unsigned char  => unsigned int
int            + short          => int           | unsigned int   + short          => unsigned int
int            + unsigned short => int           | unsigned int   + unsigned short => unsigned int
int            + int            => int           | unsigned int   + int            => unsigned int
int            + unsigned int   => unsigned int  | unsigned int   + unsigned int   => unsigned int
int            + long           => long          | unsigned int   + long           => long oder unsigned long
int            + unsigned long  => unsigned long | unsigned int   + unsigned long  => unsigned long

long           + char           => long          | unsigned long  + char           => unsigned long
long           + wchar_t        => long          | unsigned long  + wchar_t        => unsigned long
long           + signed char    => long          | unsigned long  + signed char    => unsigned long
long           + unsigned char  => long          | unsigned long  + unsigned char  => unsigned long
long           + short          => long          | unsigned long  + short          => unsigned long
long           + unsigned short => long          | unsigned long  + unsigned short => unsigned long
long           + int            => long          | unsigned long  + int            => unsigned long
long           + unsigned int   => unsigned long | unsigned long  + unsigned int   => unsigned long
long           + long           => long          | unsigned long  + long           => unsigned long
long           + unsigned long  => unsigned long | unsigned long  + unsigned long  => unsigned long
```

Zugegebenermaßen wirkt dies erst einmal erschlagend, aber es ist eigentlich nicht schwer zu begreifen. Bei jeder Rechenoperation hat jeder der 2 Operanden, sowie das Ergebnis der Rechnung einen Datentyp:

```
1. include <iostream>


int main(){
    char  Zahl1=22;
    short Zahl2=40;
    std::cout << Zahl1 * Zahl2 << std::endl; // 22   * 40    =  880
                                             // char + short => int
}
```

10.2 Gleitkommarechnen

Beim Rechnen mit Gleitkommazahlen gelten im Grunde die gleichen Regeln wie bei Ganzzahlen. Der Ergbnistyp entspricht auch hier dem des Operanden mit dem „größeren“ Typ. Die Reihenfolge lautet: `float`, `double`, `long double`. Es gilt also:

```
float       + float       => float
float       + double      => double
float       + long double => long double

double      + float       => double
double      + double      => double
double      + long double => long double

long double + float       => long double
long double + double      => long double
long double + long double => long double
```

10.3 Casting

Casting meint in diesem Zusammenhang, die Umwandlung eines Datentyps in einen anderen. Diese Typumwandlung kann sowohl automatisch (implizit) stattfinden, als auch vom Programmierer angegeben (explizit) werden.

10.3.1 Implizite Typumwandlung

Mit impliziter Typumwandlung hatten Sie bereits reichlich zu tun, denn es kann ausschließlich mit Zahlen gerechnet werden die den gleichen Typ besitzen.

Beispiele:

```
char  + int          => int          | int          + int          => int
short + unsigned int => unsigned int | unsigned int + unsigned int => unsigned int
float + double       => double       | double       + double       => double
```

Umformungsregeln

Viele binäre Operatoren, die arithmetische oder Aufzählungsoperanden erwarten, verursachen Umwandlungen und ergeben Ergebnistypen auf ähnliche Weise. Der Zweck ist, einen gemeinsamen Typ zu finden, der auch der Ergebnistyp ist. Dieses

Muster wird "die üblichen arithmetischen Umwandlungen" genannt, die folgendermaßen definiert sind:

- Wenn ein Operand vom Typ `long double` ist, dann wird der andere zu `long double` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `double` ist, dann wird der andere zu `double` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `float` ist, dann wird der andere zu `float` konvertiert.
- Ansonsten werden die integralen Umwandlungen auf beide Operanden angewendet:
- Wenn ein Operand vom Typ `unsigned long` ist, dann wird der andere zu `unsigned long` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `long` und der andere vom Typ `unsigned int`, dann wird, falls ein `long` alle Werte eines `unsigned int` darstellen kann, der `unsigned int`-Operand zu `long` konvertiert; andernfalls werden beide Operanden zu `unsigned long` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `long` ist, dann wird der andere zu `long` konvertiert.
- Andernfalls, wenn ein Operand vom Typ `unsigned int` ist, dann wird der andere zu `unsigned int` konvertiert.

Hinweis: Der einzig verbleibende Fall ist, dass beide Operanden vom Typ int sind.

Diese Regeln wurden so aufgestellt, dass dabei stets ein Datentyp in einen anderen Datentyp mit "größerem" Wertebereich umgewandelt wird. Das stellt sicher, dass bei der Typumwandlung keine Wertverluste durch Überläufe entstehen. Es können allerdings bei der Umwandlung von Ganzzahlen in `float`-Werte Rundungsfehler auftreten:

```
std::printf( "17000000 + 1.0f = %f.\n", 17000000 + 1.0f );
```

Diese Programmzeile gibt auf einem Computer, wo `float` eine 32-Bit-Gleitkommazahl ist, folgendes aus:

```
17000000 + 1.0f = 17000000.000000.
```

Warum? Für die Berechnung werden beide Operanden in den Datentyp `float` konvertiert und anschließend addiert. Eine 32-Bit-Gleitkommazahl ist aber nicht

in der Lage, Zahlen in der Größenordnung von 17 Mio. mit der nötigen Genauigkeit zu speichern, um zwischen 17000000 und 17000001 zu unterscheiden. Das Ergebnis der Addition wird daher wieder auf 17000000 gerundet. Dies geschieht in diesem Falle sogar, obwohl der Wert anschließend auf `double` "aufgeweitet" wird, da die `printf`-Methode Gleitkommawerte standardmäßig als `double` erhält (und durch die Formatangabe `"%f"` auch als `double` erwartet).

10.3.2 Explizite Typumwandlung

In C++ gibt es dafür zwei Möglichkeiten. Zum einen den aus C übernommenen Cast `(Typ)Wert` und zum anderen die vier (neuen) C++ Casts.

```
static_cast< Zieltyp >(Variable)
const_cast< Zieltyp >(Variable)
dynamic_cast< Zieltyp >(Variable)
reinterpret_cast< Zieltyp >(Variable)
```

Tip
Die Leerzeichen zwischen dem Zieltyp und den spitzen Klammern sind nicht zwingend erforderlich, Sie sollten sich diese Notation jedoch angewöhnen. Speziell wenn Sie später mit Templates oder Namensräumen arbeiten, ist es nützlich Datentypen ein wenig von ihrer Umgebung zu isolieren. Sie werden an den entsprechenden Stellen noch auf die ansonsten möglichen Doppeldeutigkeiten hingewiesen.

Thema wird Später näher erläutert...
Im Moment benötigen Sie nur den `static_cast`. Was genau die Unterschiede zwischen diesen Casts sind und wann man welchen einsetzt erfahren Sie in einem späteren Kapitel. Die C-Casts sollten Sie aber auf keinen Fall einsetzen, weshalb erfahren Sie ebenfalls später.

10.4 Ganzzahlen und Gleitkommazahlen

Wird mit einer Ganzzahl und einer Gleitkommazahl gerechnet, so ist das Ergebnis vom gleichen Typ wie die Gleitkommazahl.

10.5 Rechnen mit Zeichen

Mit Zeichen zu rechnen ist besonders praktisch. Um beispielsweise das gesamte Alphabet auszugeben zählen Sie einfach vom Buchstaben `'A'` bis einschließlich `'Z'`:

```
1. include <iostream>
```

```
int main(){
    for(char i = 'A'; i <= 'Z'; ++i){
        std::cout << i;
    }
}
```

Ausgabe

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Für eine Erklärung des obigen Quellcodes, lesen Sie bitte das Kapitel Schleifen.

Wenn Sie binäre Operatoren auf Zeichen anwenden ist das Ergebnis (mindestens) vom Typ `int`. Im folgenden Beispiel wird statt eines Buchstabens, der dazugehörige ASCII-Wert ausgeben. Um also wieder ein Zeichen auszugeben, müssen Sie das Ergebnis wieder in den Zeichentyp *casten*. (Beachten Sie im folgenden Beispiel, dass die Variable `i` - im Gegensatz zum vorherigen Beispiel - nicht vom Typ `char` ist):

```
1. include <iostream>


int main(){
    char zeichen = 'A';
    for(int i = 0; i < 26; ++i){
        std::cout << zeichen + i << ' ';              // Ergebnis int
    }
    std::cout << std::endl;
    for(int i = 0; i < 26; ++i){
        std::cout << static_cast< char >(zeichen + i); // Ergebnis char
    }
}
```

Ausgabe

```
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Kapitel 11

Verzweigungen

Eine Verzweigung (bedingte Anweisung, *conditional statement*) dient dazu, ein Programm in mehrere Pfade aufzuteilen. Beispielsweise kann so auf Eingaben des Benutzers reagiert werden. Je nachdem, was der Benutzer eingibt, ändert sich der Programmablauf.

11.1 Falls

Verzweigungen werden mit dem Schlüsselwort `if` begonnen. In der einfachsten Form sieht das so aus:

```
if(«Bedingung») «Anweisung»
```

Wenn die `Bedingung` erfüllt ist, wird die `Anweisung` ausgeführt, ansonsten wird sie übersprungen. Sollen nicht nur eine, sondern mehrere Anweisungen ausgeführt werden, fassen Sie diese mit `{...}` zu einer Blockanweisung zusammen:

```
if(«Bedingung»){
    «Anweisungen»
}
```

Als `Bedingung` darf jeder Ausdruck verwendet werden, der einen `bool` zurückgibt oder dessen Ergebnis sich in einen `bool` umwandeln lässt. Ganzzahlige und Gleit-

kommadatentypen lassen sich nach `bool` umwandeln, die Regel lautet einfach: Ist eine Zahl gleich 0 so wird sie als `false` ausgewertet, andernfalls als `true`.

```
int i;
cin >> i;
if(i){
    cout << "Der Benutzer hat einen Wert ungleich 0 eingegeben\n";
}
```

11.2 Andernfalls

Das Schlüsselwort `else` erweitert die Einsatzmöglichkeiten der Verzweigung. Während ein normales (also einzelnes) `if` einen bestimmten Teil des Codes ausführt, falls eine Bedingung erfüllt ist, stellt `else` eine Erweiterung dar, anderen Code auszuführen, falls die Bedingung nicht erfüllt ist.

```
int i;
cin >> i;
if (i)
    cout << "Sie haben einen Wert ungleich 0 eingegeben!\n";
else
    cout << "Sie haben 0 eingegeben!\n";
```

Natürlich könnten auch hier, sowohl für die `if`-Anweisung, als auch für die `else`-Anweisung, ein Anweisungsblock stehen. Wenn Sie Pascal oder eine ähnliche Programmiersprache kennen, wird Ihnen auffallen, dass auch die Anweisung vor dem `else` mit einem Semikolon abgeschlossen wird. Da auf eine `if`- oder `else`-Anweisung immer nur eine Anweisung *oder* ein Anweisungsblock stehen kann, muss zwangsläufig direkt danach ein `else` stehen, um dem `if` zugeordnet zu werden.

Sie können in einer Verzweigungsanweisung auch mehr als zwei Alternativen angeben:

```
int i;
cin >> i;
```

```
if (i == 10)
    cout << "Sie haben zehn eingegeben\n";
else
    if (i == 11)
        cout << "Sie haben elf eingegeben\n";
    else
        cout << "Sie haben weder zehn noch elf eingegeben\n";
```

Es können beliebig viele Zweige mit `else if` vorkommen. Allerdings ist es üblich, eine andere Einrückung zu wählen, wenn solche „`if-else`-Bäume“ ausgebaut werden:

```
int i;
cin >> i;
if (i == 10)
    cout << "Sie haben zehn eingegeben\n";
else if (i == 11)
    cout << "Sie haben elf eingegeben\n";
else
    cout << "Sie haben weder zehn noch elf eingegeben\n";
```

Außerdem ist es zu empfehlen, auch bei einer Anweisung einen Anweisungsblock zu benutzen. Letztlich ist die Funktionalität immer die gleiche, aber solche Blöcke erhöhen die Übersichtlichkeit und wenn Sie später mehrere Anweisungen, statt nur einer angeben möchten, brauchen Sie sich um etwaige Klammern keine Gedanken zu machen, weil sie sowieso schon vorhanden sind.

```
int i;
cin >> i;
if (i == 10) {
    cout << "Sie haben zehn eingegeben\n";
} else if(i == 11) {
    cout << "Sie haben elf eingegeben\n";
} else {
    cout << "Sie haben weder zehn noch elf eingegeben\n";
}
```

Sie werden für die Positionierung der Klammern übrigens auch oft auf eine andere Variante treffen:

```
int i;
cin >> i;
if (i == 10)
{
    cout << "Sie haben zehn eingegeben\n";
}
else
{
    if (i == 11)
    {
        cout << "Sie haben elf eingegeben\n";
    }
    else
    {
        cout << "Sie haben weder zehn noch elf eingegeben\n";
    }
}
```

Einige Programmierer finden dies übersichtlicher, für dieses Buch wurde jedoch die Variante mit den öffnenden Klammern ohne Extrazeile zu verwenden. Das hat den Vorteil, dass weniger Platz benötigt wird und da die Einrückung ohnehin die Zugehörigkeit andeutet, ist eine zusätzliche Kennzeichnung nicht unbedingt nötig.

Die Einrückung von Quelltextzeilen hat für den Compiler übrigens keine Bedeutung. Sie ist lediglich eine grafische Darstellungshilfe für den Programmierer. Auch die in diesem Buch gewählte Einrückungstiefe von vier Leerzeichen ist optional, viele Programmierer verwenden etwa nur zwei Leerzeichen. Andere hingegen sind davon überzeugt, dass acht die ideale Wahl ist. Aber egal, wofür Sie sich entscheiden, wichtig ist, dass Sie Ihren Stil einhalten und nicht ständig ihren Stil wechseln. Das verwirrt nicht nur, sondern sieht auch nicht schön aus.

Tip
Wenn es Sie nicht stört, in Ihrem Texteditor Tabulatorzeichen und Leerzeichen anzeigen zu lassen, dann sollten Sie für die Einrückung Tabulatorzeichen verwenden und für alles hinter der normalen Einrückung (etwa den Abstand bis zum Kommentar) Leerzeichen. Das hat den Vorteil, dass Sie die Einrückungstiefe jederzeit ändern können, indem Sie angeben wie viele Leerzeichen einem Tabulatorzeichen entsprechen.

11.3 Vergleichsoperatoren

Im obigen Beispiel kam schon der Vergleichsoperator `==` zum Einsatz. In C++ gibt es insgesamt sechs Vergleichsoperatoren. Sie liefern jeweils den Wert `true`, wenn die beiden Operanden (die links und rechts des Operators stehen) dem Vergleichskriterium genügen, ansonsten den Wert `false`.

`==`	identisch
`<=`	ist kleiner (oder) gleich
`>=`	ist größer (oder) gleich
`<`	ist kleiner
`>`	ist größer
`!=`	ist ungleich

Hinweis

Der Vergleichsoperator == wird von Anfängern oft mit dem Zuweisungsoperator = verwechselt. Da es absolut legal ist, eine Zuweisung innerhalb einer `if`-Bedingung zu machen, führt das oft zu schwer zu findenden Fehlern:

```
int a = 5, b = 8;
if (a = b) cout << "5 ist gleich 8.";
```

Ausgabe

```
5 ist gleich 8.
```

Das weist `b` an `a` (`a = 8`) zu und wertet dann die Rückgabe von `a` (also `8`) zu `true` aus. Prüfen Sie bei seltsamen Verhalten also immer ob vielleicht der Zuweisungsoperator = statt des Gleichheitsoperators == verwendet wurde.
Eine weitere Falle ist der Ungleichheitsoperator `!=`, wenn er falsch herum geschrieben wird (`=!`). Letzteres sind in Wahrheit zwei Operatoren, nämlich die Zuweisung = und die logische Negierung `!`, die Sie gleich kennen lernen werden. Um das zu unterscheiden, machen Sie sich einfach klar, was das in Worten heißt:

- `!=` – nicht gleich
- `=!` – gleich nicht

11.4 Logische Operatoren

Mit logischen Operatoren können Sie mehrere Bedingungen zu einem Ausdruck verknüpfen. C++ bietet folgende Möglichkeiten:

!	Logisches Nicht	Resultat wahr, wenn der Operand falsch ist
&&	Logsiches Und	Resultat wahr, wenn beide Operanden wahr sind
||	Logisches Oder	Resultat wahr, wenn mindestens ein Operand wahr ist (*inclusive-or*)

Bei den Operatoren `&&` (Logik-und) und `||` (Logik-oder) werden die Teilausdrücke *von links nach rechts* bewertet, und zwar nur so lange, bis das Resultat feststeht. Wenn z. ß;B. bei einer `&&`-Verknüpfung schon die erste Bedingung

falsch ist, wird die zweite nicht mehr untersucht, da beide Bedingungen `true` sein müssen. Der Rückgabewert dieser Operatoren ist genau wie bei den Vergleichsoperatoren von Typ `bool`.

```
int i = 10, j = 20;
if (i == 10 && j == 10) {
    cout << "Beide Werte sind gleich zehn\n";
}
if (i == 10 || j == 10) {
    cout << "Ein Wert oder beide Werte sind gleich zehn\n";
}
```

Ausgabe

```
Ein Wert oder beide Werte sind gleich zehn
```

Aus Gründen der Lesbarkeit sollten Vergleichsausdrücke grundsätzlich von Klammern umgeben sein. Der obige Code würde folglich so aussehen:

```
int i = 10, j = 20;
if ((i == 10) && (j == 10)) {
    cout << "Beide Werte sind gleich zehn\n";
}
if ((i == 10) || (j == 10)) {
    cout << "Ein Wert oder beide Werte sind gleich zehn\n";
}
```

Ausgabe

```
Ein Wert oder beide Werte sind gleich zehn
```

Hinweis

Beachten Sie bitte, dass der UND-Operator (`&&`) eine höhere Priorität als der ODER-Operator (`||`) hat. Das heißt, Sie müssen bei Ausdrücken wie dem Folgenden vorsichtig sein.

```
int i = 10, j = 20;
// Erwartete Reihenfolge    ((((i == 10) || (j == 20)) && (j == 20)) && (i == 5))
// Tatsächliche Reihenfolge ((i == 10) || (((j == 20) && (j == 20)) && (i == 5)))
if (i == 10 || j == 20 && j == 20 && i == 5) {
    cout << "i ist Zehn und fünf oder (j ist Zwanzig und i fünf)!\n";
} else {
    cout << "i ist nicht Zehn oder (j ist Zwanzig oder i nicht fünf)!\n";
}
```

Ausgabe

```
i ist Zehn und fünf oder (j ist Zwanzig und i fünf)!
```

Die Ausgabe ist von der Logik her falsch, weil der Ausdruck in der Reihenfolge `((i == 10) || (((j == 20) && (j == 20)) && (i == 5)))` ausgewertet wird. Solche Fehler sind sehr schwer zu finden, also sollten Sie sie auch nicht machen. Daher der Tipp: Verwenden Sie bei solch komplexen Bedingungen *immer* Klammern, um klar zu machen, in welcher Reihenfolge Sie die Ausdrücke auswerten wollen. Das ist unmissverständlich, und der menschlicher Leser liest den Ausdruck genauso wie der Compiler. Um zu erkennen, an welcher Stelle eine Klammer wieder geschlossen wird, beherrschen die meisten Editoren das sogenannte Bracket Matching. Dabei hebt der Editor (automatisch oder über einen bestimmten Hotkey) die schließende Klammer hervor.

Die Operatoren lassen sich übersichtlich mit Wahrheitstafeln beschreiben:

Logisches Und				
a	true	true	false	false
b	true	false	true	false
a && b	true	false	false	false

Logisches Oder				
a	true	true	false	false
b	true	false	true	false
a || b	true	true	true	false

Logisches Nicht		
`a`	`true`	`false`
`!a`	`false`	`true`

Tip
Wenn Sie mit mehreren `&&` und `||` arbeiten, dann schreiben Sie den Ausdruck, der am wahrscheinlichsten zutrifft, auch am weitesten links, also **vor** den anderen Ausdrücken. Das ist eine (zugegebenermaßen) sehr geringfügige Optimierung, aber es gibt Situationen in denen Sie trotzdem sinnvoll ist. Beispielsweise wenn die Bedingung innerhalb einer Schleife (siehe Kapitel Schleifen) sehr oft ausgeführt wird. Hinweis für fortgeschrittene Leser: Beachten Sie bitte, dass für überladende Operatoren andere Regeln gelten. Alle diejenigen, die noch nicht wissen, was überladende Operatoren sind, brauchen sich um diesen Hinweis (noch) nicht zu kümmern.

11.5 Bedingter Ausdruck

Häufig werden Verzweigungen eingesetzt, um abhängig vom Wert eines Ausdrucks eine Zuweisung vorzunehmen. Das können Sie mit dem Auswahloperator `? ... : ...` auch einfacher formulieren:

```
min = a < b ? a : b;
// Alternativ ginge:
if (a < b) {
    min = a;
} else {
    min = b;
}
```

Grafisch sieht das so aus:

```
      =
Ziel/   \Quelle
  min   a<b
     ja/   \nein
      a     b
```

Der Variablen `min` wird der kleinere, der beiden Werte `a` und `b` zugewiesen. Analog zum Verhalten der logischen Operatoren wird nur derjenige „Zweig“ bewertet, der nach Auswertung der Bedingung (`a < b`) tatsächlich ausgeführt wird.

Kapitel 12

Schleifen

Mit dem, was Sie bis jetzt gelernt haben, sollte es für Sie eine leichte Übung sein, die Zahlen von eins bis zehn ausgeben zu lassen. So könnte ein Programm aussehen, das dies tut:

```
    1. include <iostream>


using namespace std;
int main() {
    cout << "1\n";
    cout << "2\n";
    cout << "3\n";
    cout << "4\n";
    cout << "5\n";
    cout << "6\n";
    cout << "7\n";
    cout << "8\n";
    cout << "9\n";
    cout << "10\n";
}
```

Ausgabe

```
1
2
3
4
5
6
7
```

```
8
9
10
```

Dieses Programm ist einfach – aber was wäre, wenn die Zahlen eins bis einer Million ausgegeben werden sollen? Oder schlimmer noch - Ja, es geht noch schlimmer - die Ausgabe hängt von der Benutzereingabe ab. Dann müssten Sie von der größtmöglichen Zahl ausgehen (bei `unsigned int` üblicherweise 4.294.967.295) und auch noch nach jeder Eingabe überprüfen, ob die vom Benutzer eingegebene Zahl erreicht ist.

```
    1. include <iostream>


using namespace std;
int main() {
    unsigned int i=1, benutzer;  // 2 Variablen anlegen
    cin >> benutzer;             // Benutzer gibt Zahl ein
    if (i <= benutzer) {         // Benutzereingabe erreicht?
        cout << "1\n";           // Ausgabe der Zahl 1
        ++i;                     // Ausgegebene Zahlen mitzählen
    } else return 0;             // Anwendung beenden
    if (i <= benutzer) {         // Benutzereingabe erreicht?
        cout << "2\n";           // Ausgabe der Zahl 2
        ++i;                     // Ausgegebene Zahlen mitzählen
    } else return 0;             // Anwendung beenden
    // ...
    if (i <= benutzer) {         // Benutzereingabe erreicht?
        cout << "4294967295\n";  // Ausgabe der Zahl 4294967295
        ++i;                     // Ausgegebene Zahlen mitzählen
    } else return 0;             // Anwendung beenden
    // Wenn Ihr Compiler diese Stelle erreichen soll, brauchen Sie einen leistungsstarken
    // Rechner und ein robustes Betriebssystem.
    // Um dieses Problem zu lösen, setzen Sie sich einfach in die nächste Zeitmaschine
    // und bestellen Sie sich einen Rechner aus dem Jahr 2020!
}
```

Es sei Ihnen überlassen, den fehlenden Code von Hand einzutragen und wenn Sie fertig sind können Sie mit Stolz behaupten, dass Sie ein simples Problem mit einer Anzahl von über 17,1 *Milliarden* Zeilen Code gelöst haben. Die GCC hat gerade einmal etwas mehr als 2,1 *Millionen* Zeilen Code, aber da geht es ja

auch nur darum, Compiler zu schreiben und nicht Zahlen auszugeben. Trotzdem, die Unterschiede zwischen dem Compilerbau-Projekt und unserem hoch komplexen Programm zum Ausgeben einer durch den Benutzer angegebenen Menge von Zahlen ist dermaßen horrend, dass es da doch eigentlich noch einen Trick geben muss.

Und tatsächlich, denn an diesem Punkt kommen Schleifen ins Spiel. Bis jetzt wurden alle Programme einfach der Reihe nach abgearbeitet und zwischendurch wurde eventuell mal eine Abzweigung genommen. Mit einer Schleife können Sie erreichen, dass ein Programmteil mehrfach abgearbeitet wird. C++ stellt 3 Schleifenkonstrukte zur Verfügung. Die kopfgesteuerte `while`-Schleife, die fußgesteuerte `do-while`-Schleife und die ziemlich universelle (ebenfalls kopfgesteuerte) `for`-Schleife. Was kopf- und fußgesteuerte Schleife bedeutet erfahren Sie in Kürze. Vielleicht wissen Sie es aber auch schon aus dem Kapitel „Grundlegende Elemente“, aus dem Abschnitt „Für Programmieranfänger“.

12.1 Die `while`-Schleife

Eine `while`-Schleife hat die folgende allgemeine Form:

```
while(«Bedingung») «Anweisung»
```

Natürlich können Sie, wie bei den Verzweigungen auch, hier wieder mehrere Anweisungen zu einem Anweisungsblock zusammenfassen, da diese als eine Anweisung gilt:

```
while(«Bedingung»){
    «Anweisungen»
}
```

Solange die Bedingung erfüllt ist, wird die Anweisung oder der Anweisungsblock ausgeführt. Da es sich hier um eine kopfgesteuerte Schleife handelt, wird erst die Bedingung ausgewertet. Ist diese erfüllt, so wird die Anweisung ausgeführt und dann erneut die Bedingung überprüft. Ist die Bedingung nicht erfüllt, wird der Schleifeninhalt kurzerhand übersprungen und mit dem Quelltext nach der Schleife fortgesetzt. Was eine fußgesteuerte Schleife macht, erfahren Sie unter der Über-

schrift `do-while`. Eine Gegenüberstellung der beiden Schleifen gibt es in der Zusammenfassung dieses Kapitels.

Tip

Wie die Bedingungen ausgewertet werden, können Sie im Kapitel „Verzweigungen“ nachlesen. Einige von Ihnen werden sich vielleicht noch daran erinnern, dass dies das vorherige Kapitel war, aber für die Gedächtnisschwachen unter Ihnen und jene die es einfach nicht lassen können in der Mitte mit dem Lesen zu beginnen, steht es hier noch mal Schwarz auf Grün.

Hinweis

Beachten Sie, dass Schleifen so lange ausgeführt werden, bis die Bedingung nicht mehr erfüllt ist. Wenn Sie also nicht innerhalb der Schleife dafür sorgen, dass die Bedingung irgendwann nicht mehr erfüllt ist, dann haben Sie eine sogenannte *Endlosschleife*. Das heißt, der *Schleifenrumpf* (so nennt man die Anweisung oder den Anweisungsblock einer Schleife) wird immer wieder ausgeführt. Das Programm wird also nie beendet. In Kombination mit einem Schlüsselwort, das Sie in Kürze kennen lernen werden, kann eine solche Endlosschleife durchaus gewollt und sinnvoll sein, aber in der Regel entsteht so etwas versehentlich. Wenn Ihr Programm also mal „abgestützt“ ist, im Sinne von „Es reagiert nicht mehr! Wie schrecklich!“, dann haben Sie fast todsicher irgendwo eine Endlosschleife fabriziert.

Nun aber zurück zu unserer Schleifenaufgabe mit 17,1 Milliarden Zeilen Code. Da Ihr Compiler immer noch nicht damit fertig ist, die oben vorgestellte Lösung zu übersetzen, versuchen wir jetzt mal das ganze mit einer `while`-Schleife zu lösen.

```
    1. include <iostream>


using namespace std;
int main() {
    unsigned int i=1, benutzer;  // 2 Variablen anlegen
    cin >> benutzer;             // Benutzer gibt Zahl ein
    while (i <= benutzer) {      // Benutzereingabe erreicht?
        cout << i << endl;       // Ausgabe von i und einem Zeilenumbruch
        i++;                     // i um eins erhöhen (-> ausgegebene Zahlen mitzählen)
    }
}
```

Nun ja, das sieht dem Programm von oben doch irgenwie ähnlich, nur die knapp 4,3 Milliarden `if`-Anweisungen sind weggefallen und haben einer vom Aufbau fast identischen `while`-Schleife platz gemacht. Nun haben wir natürlich ein Problem: Ihr Superrechner aus dem Jahr 2020 ist immer noch mit der Übersetzung der ersten Programmversion beschäftigt und wird es wohl auch noch bis zu seiner Erfindung bleiben. Aber zum Glück haben Sie ja noch einen alten Rechner von 1980. Also versuchen Sie das Programm auf diesem zu übersetzen und auszuführen. Tatsächlich. Es funktioniert. Erstaunlich, dass so eine alte Kiste einen Rechner überholt, den Sie sich extra aus der Zukunft haben liefern lassen, um die maximale Leistungsstärke zu bekommen.

Wenn Sie Schwierigkeiten haben das Beispiel nachzuvollziehen, dann sehen Sie sich noch mal die Schleifen-Version für die Zahlen von 1 bis 10 an.

```
    1. include <iostream>


using namespace std;
int main() {
    int i = 1;                // Anlegen von i
    while (i <= 10) {         // Ist i noch kleiner-gleich 10?
        cout << i << endl;    // Ausgabe von i und neue Zeile
        i++;                  // i um eins erhöhen
    }
}
```

Ausgabe

```
1
2
3
4
5
6
7
8
9
10
```

So sieht das Ganze schon viel kompakter und vielleicht auch übersichtlicher aus als die Version von ganz oben. Die Ausgabe dagegen ist völlig identisch. Hier wird `i` am Anfang auf eins gesetzt. Somit ist die Bedingung (eins ist kleiner oder gleich zehn) erfüllt. Damit wird der Schleifenrumpf ausgeführt. „1“ wird ausgegeben. Es folgt ein Zeilenumbruch. Danach wird der Wert von `i` um eins erhöht. Danach

wird wieder geprüft, ob die Bedingung erfüllt ist. Da `i` jetzt zwei ist, lautet die Bedingung „zwei ist kleiner oder gleich zehn“, da dies eine wahre Aussage ist, wird wieder der Schleifenrumpf ausgeführt. Das wiederholt sich bis `i` Schließlich den Wert elf hat. Die Bedingung lautet dann „elf ist kleiner oder gleich zehn“, diese Aussage ist zweifellos falsch. Daher wird der Schleifenrumpf nun übersprungen und mit dem Code dahinter weitergemacht. Da in unserem Beispiel dort aber kein Code mehr folgt, wird das Programm beendet.

Zusammengefasst:

1: Quelltext vor der Schleife
2: Schleifen Bedingung

- Erfüllt:

  2.1: Schleifenrumpf
  2.2: weiter mit 2

- Nicht Erfüllt:

  2.1: weiter mit 3

3: Quelltext nach der Schleife

12.2 Die `do-while`-Schleife

Wie versprochen lüften wir nun das Geheimnis um die fußgesteuerten Schleifen. `do-while` ist eine fußgesteuerte Schleife, dass heißt, als erstes wird der Schleifenrumpf ausgeführt, danach die Bedingung überprüft und dann abhängig von der Bedingung, wieder der Rumpf ausgeführt (Bedingung erfüllt) oder mit dem Quelltext nach der Schleife fortgesetzt (Bedingung nicht erfüllt). Eine fußgesteuerte Schleife zeichnet sich also dadurch aus, dass der Schleifenrumpf mindestens ein mal ausgeführt wird.

```
do «Anweisung» while(«Bedingung»);
```

Bei dieser Schleife finden wir die obige Syntax nur selten, da der Schleifenrumpf hier zwischen den beiden Schlüsselwörtern `do` und `while` steht, wird fast immer ein Anweisungsblock benutzt, auch wenn nur eine Anweisung vorhanden ist. Für Ihren Compiler spielt das natürlich keine Rolle, aber für einen Menschen der den Quelltext liest ist es übersichtlicher.

```
do{
    «Anweisungen»
}while(«Bedingung»);
```

Unser anfängliches Riesenprogramm sieht mit einer `do-while` Schleife so aus:

```
   1. include <iostream>


using namespace std;
int main() {
    unsigned int i=1, benutzer;  // 2 Variablen anlegen
    cin >> benutzer;             // Benutzer gibt Zahl ein
    do {                         // Schleifenanfang
        cout << i << endl;       // Ausgabe von i
        ++i;                     // Ausgegebene Zahlen mitzählen
    } while (i <= benutzer);     // Benutzereingabe erreicht?
}
```

Sie werden feststellen das die Ausgabe dieses Programms mit der Ausgabe in der `while`-Schleifen Version übereinstimmt. Den Unterschied bemerken Sie, wenn

Sie 0 eingeben, während die `while`-Version keine Ausgabe macht, gibt diese `do-while`-Version „1“ und einen Zeilenumbruch aus, denn der Schleifenrumpf wird im erst einmal ausgeführt, erst danach wird die Bedingung überprüft und entschieden ob er noch einmal ausgeführt werden muss.

Zusammengefasst:

1: Quelltext vor der Schleife
2: Schleifenrumpf
3: Schleifen Bedingung

- Erfüllt:

  3.1: weiter mit 2

- Nicht Erfüllt:

  3.1: weiter mit 4

4: Quelltext nach der Schleife

12.3 Die `for`-Schleife

Die `for`-Schleife ist etwas komplexer als die vorherigen beiden Schleifen. Sie gliedert sich in Teile:

```
for(«Initialisierungsteil»; «Bedingungsteil»; «Anweisungsteil») «Schleifenrumpf»
```

Um etwas Platz zu sparen wurde in dieser Beschreibung einfach Schleifenrumpf geschrieben, anstatt wieder einzeln aufzuführen das sowohl eine einzelne (durch Semikolon abgeschlossene) Anweisung, als auch ein Anweisungsblock möglich sind. Der Bedingungsteil verhält sich genau wie bei der `while`- und der `do-while`-Schleife oder sagen wir fast genau so, denn einen kleinen aber feinen Unterschied gibt es doch. Während `while` und `do-while` immer eine Bedingung erwarten, muss bei einer `for`-Schleife nicht unbedingt eine Bedingung angegeben werden. Wenn keine Bedingung angegeben ist, wird einfach angenommen das die Bedingung immer erfüllt ist, Sie erhalten eine Endlosschleife. Wie das sinnvoll eingesetzt wird erfahren Sie in kürze.

Im Anweisungsteil können Sie eine beliebige Anweisung ausführen, dieser Teil wird oft verwendet, um Variablen bei jedem Schleifendurchlauf hoch oder runter zu zählen. Im nächsten Beispiel wird dies auch demonstriert. Es ist auch möglich,

mehrere solcher „hoch- oder runter-Zähl“-Anweisungen durch Komma getrennt anzugeben, das wird im nächstem Beispiel aber nicht gemacht. Der Anweisungsteil wird übrigens direkt nach dem Schleifenrumpf und vor dem nächsten Bedingungstest ausgeführt. Der Initialisierungsteil ist dem Anweisungsteil dahingehend ähnlich, als dass auch hier eine beliebige Anweisung ausgeführt werden kann. Zusätzlich ist es hier aber noch möglich Variablen eines Datentyps anzulegen. Sie können also problemlos 2 `int`-Variablen anlegen, aber nicht eine `int`- und eine `char`-Variable. Der Initialisierungsteil wird nur einmal am Beginn der Schleife ausgeführt.

Thema wird Später näher erläutert...

Im Kapitel Lebensdauer und Sichtbarkeit von Variablen werden Sie noch etwas genaueres darüber erfahren wo Sie die im Initialisierungsteil der Schleife angelegten Variablen verwenden können. Dort werden Sie auch erfahren wie eine `for`-Schleife mit Hilfe einer `while`-Schleife „nachgebaut“ werden kann.
Jetzt sollten Sie sich jedoch merken das Sie eine solche Variable, nur innerhalb der Schleife, also nicht mehr nach ihrem verlassen, verwenden können.

```
   1. include <iostream>


using namespace std;
int main() {
    unsigned int benutzer;                      // Variablen für Benutzereingabe
    cin >> benutzer;                            // Benutzer gibt Zahl ein
    for(unsigned int i = 1; i <= benutzer; ++i) // for-Schleife
        cout << i << endl;                      // Ausgabe von i
}
```

Zusammengefasst:

1: Quelltext vor der Schleife
2: Initialisierungsteil der Schleife
3: Schleifen Bedingung

- Erfüllt:

  3.1: Schleifenrumpf
  3.2: Anweisungsteil
  3.3: weiter mit 3

- Nicht Erfüllt:

  3.1: weiter mit 4

4: Quelltext nach der Schleife

12.4 Die `break`-Anweisung

Jetzt ist es an der Zeit, das Geheimnis um die sinnvolle Verwendung von Endlosschleifen zu enthüllen. Die `break`-Anweisung wird innerhalb von Schleifen verwendet, um die Schleife sofort zu beenden. Der Quelltext wird dann ganz normal nach der Schleife fortgesetzt. `break` können Sie in jeder der drei Schleifen verwenden. Das folgende Beispiel demonstriert den Einsatz von `break`, anhand unseres Lieblingsprogramms in diesem Kapitel.

```
    1. include <iostream>


using namespace std;
int main() {
    unsigned int benutzer, i=1;    // Variablen für Benutzereingabe
    cin >> benutzer;               // Benutzer gibt Zahl ein
    for(;;){                       // Endlos-for-Schleife
        cout << i << "\n";         // Ausgabe von i
        ++i;                       // Variable erhöhen
        if(i>benutzer)break;       // Abbrechen, wenn Bedingung erfüllt
    }
}
```

Sie können aus jeder Schleife eine Endlosschleife machen, hier wurde die `for`-Schleife gewählt um zu zeigen wie Sie ohne Bedingung aussieht. Bei einer `while`- oder `do-while`-Schleife könnten Sie beispielsweise `true` als Bedingung angeben. Die for-Schleife legt jetzt übrigens das gleiche Verhalten an den Tag, wie eine `do-while`-Schleife. Sie können auch problemlos nur einen oder zwei Teile des Schleifenkopfes bei `for`-Schleife übergeben, wichtig ist nur dass Sie die beiden Semikolons immer angeben, da Sie dem Compiler mitteilen, was welcher Teil des Schleifenkopfes ist.

Nun wissen Sie wieder etwas mehr über `for`, dabei sollte es unter dieser Überschrift doch eigentlich um `break` gehen. Wie Sie aber sehen, hängt letztlich alles mit allem zusammen und so ist es oft schwer eine richtige Abgrenzung zu schaffen.

12.5 Die `continue`-Anweisung

Das zweite wichtige Schlüsselwort für Schleifen ist `continue`. Es wird genau so verwendet wie `break`, bricht die Schleife allerdings nicht völlig ab, sondern setzt die Codeausführung am Endes des Schleifenrumpfes fort. Für `while` und `do-while` bedeutet das beim Bedingungstest, für die `for`-Schleife beim Anweisungsteil. Mit `continue` können Sie also den Rest des aktuellen Schleifendurchlaufs überspringen. Wir sehen uns das wieder anhand des Beispiels an.

```
    1. include <iostream>


using namespace std;
int main() {
    unsigned int benutzer;                          // Variablen für Benutzereingabe
    cin >> benutzer;                                // Benutzer gibt Zahl ein
    for(unsigned int i = 1; i <= benutzer; ++i) { // for-Schleife
        if (i%2 == 0) continue;                     // alle geraden Zahlen überspringen
        cout << i << endl;                          // Ausgabe von i
    }
}
```

Hier werden nur ungrade Zahlen ausgegeben, da der `%`-Operator (liefert den Rest einer Ganzzahligen Division) bei allen geraden Zahlen, (teilt man durch 2 ist als Rest ja nur 0 oder 1 möglich,) 0 zurückliefert und somit `continue` ausgeführt wird.

Natürlich könnten Sie das auch noch auf eine andere Weise realisieren. Aber es gibt ja beim Programmieren viele Wege, die nach Rom führen, wie in diesem Kapitel anhand der verschieden Schleifen schon bewiesen wurde. Leider gibt es aber noch mehr Wege, die auf direktem Wege an Rom vorbeiführen... Aber hier ist für das Beispiel von eben noch ein Pfad, der sicher nach Rom führt:

```
    1. include <iostream>


using namespace std;
int main() {
    unsigned int benutzer;                          // Variablen für Benutzereingabe
    cin >> benutzer;                                // Benutzer gibt Zahl ein
    for(unsigned int i = 1; i <= benutzer; i += 2) // for-Schleife mit 2er Schritten
        cout << i << endl;                          // Ausgabe von i
```

```
}
```

12.6 Kapitelanhang

Aufgaben

Aufgabe 1

Schreiben Sie mithilfe einer Schleife ein kleines Spiel. Der Spielablauf lauten:

- Am Anfang gibt der Spieler einen Zahlenbereich ein. (Zum Beispiel: 1-100)
- Der Spieler muss sich innerhalb dieses Bereiches eine Zahl merken (eingegebene Grenzzahlen sind nicht zulässig).
- Das Programm soll dann die Zahl erraten. Der Benutzer teilt dem Programm mit, ob die Zahl an die er denkt kleiner, größer oder gleich der vom Programm geratenen Zahl ist. Die kann zum Beispiel über die Eingabe von <, > und = erfolgen.

Musterlösung

Es gibt natürlich viele Wege dieses Problem zu lösen und Sie sehen ja ob Ihr Programm funktioniert oder nicht. Hier wird nur eine Musterlösung vorgestellt, falls Sie überhaupt nicht zurecht kommen oder sich einfach dafür interessieren wie der Autor an das Problem herangegangen ist.

```
1. include <iostream>


using namespace std;
int main() {
    int min, max;                                  // Variablen für den möglichen Zahlenbereich
    int zahl;                                      // Zahl die der Rechner vermutet
    cout << "Wo fängt die Zahlenreihe an?: ";      // Zahlenbereich abfragen
    cin >> min;                                    // Benutzereingabe einlesen
    cout << "Wo hört die Zahlenreihe auf?: ";      // Zahlenbereich abfragen
    cin >> max;                                    // Benutzereingabe einlesen
    for (char eingabe = '0'; eingabe != '=';) {    // Abbrechen wenn eingabe '=' ist
        zahl = min + (max - min) / 2;              // Mittlere Zahl berechnen
        cout << "Denken Sie an " << zahl << "? "; // Vermutung ausgeben
        cin >> eingabe;                            // Antwort einlesen
        if (eingabe == '<')                        // Ist die Zahl kleiner?
```

```
            max = zahl;                             // Setzte max auf den zu großen Wert zahl
        else if (eingabe == '>')                    // Ist die Zahl größer?
            min = zahl;                             // Setzte min auf den zu kleinen Wert zahl
        else if (eingabe != '=')                    // Ist Eingabe auch kein Gleichheitszeichen
            cout << "Sie haben ein unzulässiges Zeichen eingegeben!\n"; // Fehlerhafte Eingabe
 melden
        if (min+1 >= max) {                         // Keine Zahl mehr im gültigen Bereich
            cout << "Sie sind ein Lügner!\n";       // Das Programm ist äußert entsetzt
            break;                                  // Schleife wird abgebrochen
        }
    }
    cout << "Die von Ihnen gemerkte Zahl ist " << zahl << "!" << endl; // Ausgabe der erratenen
 Zahl
}
```

Ausgabe

```
Wo fängt die Zahlenreihe an?: 0
Wo hört die Zahlenreihe auf?: 100
Denken Sie an 50? <
Denken Sie an 25? >
Denken Sie an 37? <
Denken Sie an 31? >
Denken Sie an 34? <
Denken Sie an 32? >
Denken Sie an 33? =
Die von Ihnen gemerkte Zahl ist 33!
```

Kapitel 13

Auswahl

Verzweigungen kennen Sie schon, daher sollte es Ihnen nicht schwer fallen, ein Programm zu schreiben, welches abfragt, welche der verschiedenen, in der Überschrift benannten Leckereien, Sie am liebsten mögen. Das ganze könnte etwa so aussehen:

```
    1. include <iostream>


using namespace std;
int main(){
    int auswahl;
    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
            "1 - Käsesahnetorte\n"
            "2 - Streuselkuchen\n"
            "3 - Windbeutel\n";
    cin >> auswahl;
    if(auswahl==1)
        cout << "Sie mögen also Käsesahnetorte!\n";
    else if(auswahl==2)
        cout << "Streuselkuchen ist ja auch lecker...\n";
    else if(auswahl==3)
        cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher
 bestätigen?\n";
    else
        cout << "Wollen Sie wirklich behaupten, das Ihnen nichts davon zusagt?\n";
    return 0;
}
```

Ausgabe

```
Wählen Sie Ihre Lieblingsleckerei:
1 - Käsesahnetorte
2 - Streuselkuchen
3 - Windbeutel
<Eingabe>2</Eingabe>
Streuselkuchen ist ja auch lecker...
```

Das funktioniert natürlich, aber finden Sie nicht auch, dass der (Schreib-)Aufwand ein wenig zu groß ist? Außerdem ist diese Schreibweise nicht gerade sofort einleuchtend. Für unseren Fall, wäre es schöner, wenn wir die Variable `auswahl` als zu untersuchendes Objekt festlegen könnten und dann einfach alle Fälle die uns interessieren durchgehen könnten. Praktischerweise bietet C++ ein Schlüsselwort, das genau dies ermöglicht.

13.1 `switch` und `break`

Eine `switch`-Anweisung hat die folgende Form:

```
switch(«Ganzzahlige Variable»){
    case «Ganzzahl»: «Anweisungsblock»
    case «Ganzzahl»: «Anweisungsblock»
    «...»
    default: «Anweisungsblock»
}
```

Auf unser Beispiel angewandt, würde das dann so aussehen:

```
    1. include <iostream>


using namespace std;
int main(){
    int auswahl;
    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
            "1 - Käsesahnetorte\n"
            "2 - Streuselkuchen\n"
            "3 - Windbeutel\n";
    cin >> auswahl;
    switch(auswahl){
```

```
        case 1:  cout << "Sie mögen also Käsesahnetorte!";
        case 2:  cout << "Streuselkuchen ist ja auch lecker...";
        case 3:  cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher
 bestätigen?";
        default: cout << "Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?";
    }
    return 0;
}
```

Ausgabe

```
Wählen Sie Ihre Lieblingsleckerei:
1 - Käsesahnetorte
2 - Streuselkuchen
3 - Windbeutel
<Eingabe>2</Eingabe>
Streuselkuchen ist ja auch lecker...
Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher bestätigen?
Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?
```

Nun ja, Sie haben ein neues Schlüsselwort kennen gelernt, setzen es ein und kommen damit auch prompt zum falschen Ergebnis. Sieht so aus, als würde jetzt jeder der Sätze ab dem Ausgewählten ausgegeben. In der Tat handelt `switch` genau so. Es führt alle Anweisungen, ab dem Punkt aus, an dem das `case`-Argument mit der übergebenen Variable (in unserem Fall `auswahl`) übereinstimmt. Um das Ausführen nachfolgender Anweisungen innerhalb von `switch` zu verhindern, benutzen Sie das Schlüsselwort `break`. Unser Beispiel sieht dann also folgendermaßen aus:

```
    1. include <iostream>


using namespace std;
int main(){
    int auswahl;
    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
            "1 - Käsesahnetorte\n"
            "2 - Streuselkuchen\n"
            "3 - Windbeutel\n";
    cin >> auswahl;
    switch(auswahl){
        case 1:
            cout << "Sie mögen also Käsesahnetorte!";
            break;
```

```
        case 2:
            cout << "Streuselkuchen ist ja auch lecker...";
            break;
        case 3:
            cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher
 bestätigen?";
            break;
        default:
            cout << "Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?";
    }
    return 0;
}
```

Ausgabe

```
Wählen Sie Ihre Lieblingsleckerei:
1 - Käsesahnetorte
2 - Streuselkuchen
3 - Windbeutel
<Eingabe>2</Eingabe>
Streuselkuchen ist ja auch lecker...
```

Nun haben Sie wieder das ursprüngliche, gewünschte Verhalten. Was Sie noch nicht haben, ist eine Erklärung, warum extra ein `break` aufgerufen werden muss, um das Ausführen folgender `case`-Zweige zu unterbinden. Aber das werden Sie gleich erfahren.

13.2 Nur Ganzzahlen

Wie bereits aus der oben stehenden Syntaxangabe hervorgeht, kann `switch` nur mit Ganzzahlen benutzt werden. Dies hängt damit zusammen, dass `switch` *ausschließlich* auf Gleichheit mit einem der `case`-Zweige vergleicht. Bei `int`-Werten (mit der üblichen Größe von 4 Byte) ist das noch eine „überschaubare“ Menge von maximal 4.294.967.296 (2^{32}) `case`-Zweigen. Ein `float` kann die gleiche Menge unterschiedlicher Zahlen darstellen, sofern er ebenfalls 4 Byte groß ist. Aber versuchen Sie mal einen genauen `float`-Wert aufzuschreiben. Sie arbeiten also immer mit Näherungen, wenn Sie Gleitkommazahlen in Ihrem Code benutzen. Ihr Compiler regelt das für Sie, er weist der Gleitkommazahl den zu Ihrem Wert am nächsten liegenden, darstellbaren Wert zu. Deshalb ist es so gut wie nie

sinnvoll (aber natürlich trotzdem möglich), einen Test auf Gleichheit für Gleitkommazahlen durchzuführen. Bei `switch` wurde das jedoch unterbunden.

13.3 Zeichen und mehrere `case`-Zweige

Eine `if`-Anweisung trifft Ihre Entscheidung hingegen, anhand der Tatsache ob eine Bedingung erfüllt ist, oder nicht. Für die `if`-Anweisung gibt es also nur zwei Möglichkeiten. Allerdings gibt es eine ganze Menge von Möglichkeiten, wie der Zustand `true`, oder `false` erreicht werden kann. Die Rede ist von Vergleichen und Verknüpfungen (wie `&&` (= und) oder `||` (= oder)) und der Umwandlung von anderen Datentypen nach `true`, oder `false`. Für ganzzahlige Datentypen gilt etwa, dass jeder Wert ungleich 0 `true` ergibt, aber 0 hingegen wird als `false` ausgewertet. Für eine genaue Beschreibung dessen, was hier noch mal kurz umrissen wurde, sehen Sie sich das Kapitel über Verzweigungen an.

Also gut, `switch` kann nur auf Gleichheit testen und erlaubt auch keine Verknüpfungen. Oder sagen wir besser: fast keine. Denn genau deshalb gibt es diese scheinbar aufwendige Regelung mit dem `break`. Das folgende Beispiel demonstriert die Verwendung von Zeichen und die Anwendung von mehreren `case`-Zweigen, die den gleichen Code ausführen (was einer Verknüpfung mit `||` zwischen mehreren Vergleichen mittels == gleichkommt):

```
1. include <iostream>


using namespace std;
int main(){
    char auswahl;
    cout << "Wählen Sie Ihre Lieblingsleckerei:\n"
            "K - Käsesahnetorte\n"
            "S - Streuselkuchen\n"
            "W - Windbeutel\n";
    cin >> auswahl;
    switch(auswahl){
        case 'k':
        case 'K':
            cout << "Sie mögen also Käsesahnetorte!";
            break;
        case 's':
        case 'S':
            cout << "Streuselkuchen ist ja auch lecker...";
```

```
            break;
        case 'w':
        case 'W':
            cout << "Windbeutel sind so flüchtig wie ihr Name, das können Sie sicher
 bestätigen?";
            break;
        default:
            cout << "Wollen Sie wirklich behaupten, dass Ihnen nichts davon zusagt?";
    }
    return 0;
}
```

Ausgabe

```
Wählen Sie Ihre Lieblingsleckerei:
K - Käsesahnetorte
S - Streuselkuchen
W - Windbeutel
<Eingabe>s</Eingabe>
Streuselkuchen ist ja auch lecker...
```

Dieses Programm benutzt die Anfangbuchstaben anstatt der bisherigen Durchnummerierung für die Auswahl genutzt. Außerdem ist es möglich sowohl den großen, als auch den kleinen Buchstaben einzugeben. Beides führt den gleichen Code aus.

Kapitel 14

Ein Taschenrechner wird geboren

In diesem Kapitel wollen wir einen kleinen Taschenrechner für die Kommandozeile schreiben. Dieser wird in späteren Kapiteln noch verbessert, aber fürs Erste lernt er nur die vier Grundrechenarten und kann auch nur mit je 2 Zahlen rechnen.

14.1 Die Eingabe

Ein Aufgabe besteht aus zwei Zahlen, die durch ein Rechenzeichen getrennt sind. Für die Zahlen verwenden wir den Typ `double`. Das Rechenzeichen lesen wir als `char` ein. Außerdem brauchen wir noch eine Variable in der wir das Ergebnis speichern. Diese soll ebenfalls vom Typ `double` sein.

```
double zahl1, zahl2, ergebnis;
char rechenzeichen;
cin >> zahl1 >> rechenzeichen >> zahl2;
```

14.2 Die 4 Grundrechenarten

Wie sicher jeder weiß, sind die 4 Grundrechenarten Addition (`+`), Subtraktion (`-`), Multiplikation (`*`) und Division (`/`). Da die Variable `rechenzeichen` vom Typ `char` und `char` ein ganzzahliger Typ ist, bietet es sich an, die `switch`-Anweisung zu verwenden um die richtige Rechnung zu ermitteln und durchzuführen. Wenn

ein ungültiges Rechenzeichen eingegeben wird, geben wir eine Fehlermeldung aus und beenden das Programm.

```
switch(rechenzeichen){
    case '+': ergebnis = zahl1+zahl2; break;
    case '-': ergebnis = zahl1-zahl2; break;
    case '*': ergebnis = zahl1*zahl2; break;
    case '/': ergebnis = zahl1/zahl2; break;
    default: cout << "unbekanntes Rechenzeichen...\n"; return 1;
}
```

Der Rückgabewert `1` - ausgelöst von `return 1;` - beendet das Programm, die `1` (und jeder andere Wert ungleich `0` auch) sagt dabei aus, dass im Programm ein Fehler auftrat.

14.3 Das ganze Programm

Hier ist noch einmal eine Zusammenfassung unseres Taschenrechners:

```
1. include <iostream>


using namespace std;
int main(){
    double zahl1, zahl2, ergebnis;               // Variablen für Zahlen
    char rechenzeichen;                          // Variable fürs Rechenzeichen
    cout << "Geben Sie eine Rechenaufgabe ein: "; // Eingabeaufforderung ausgeben
    cin >> zahl1 >> rechenzeichen >> zahl2;      // Aufgabe einlesen
    switch(rechenzeichen){                       // Wert von rechenzeichen ermitteln
        case '+': ergebnis = zahl1+zahl2; break;  // entsprechend dem
        case '-': ergebnis = zahl1-zahl2; break;  // Rechenzeichen
        case '*': ergebnis = zahl1*zahl2; break;  // das Ergebnis
        case '/': ergebnis = zahl1/zahl2; break;  // berechnen
        // Fehlerausgabe und Programm beenden, falls falsches Rechenzeichen eingegeben wurde
        default: cout << "unbekanntes Rechenzeichen...\n"; return 1;
    }
    // Aufgabe noch mal komplett ausgeben
    cout << zahl1 << ' ' << rechenzeichen << ' ' << zahl2 << " = " << ergebnis << '\n';
}
```

Ausgabe

```
Geben Sie eine Rechenaufgabe ein: <Eingabe>99 / 3</Eingabe>
99 / 3 = 33
```

Kapitel 15

Zusammenfassung

Ein C++-Programm beginnt mit der Hauptfunktion `main()`.

15.1 Ein- und Ausgabe

Die Ein- und Ausgabe erfolgt in C++ über Streams, mehr dazu erfahren Sie in einem der folgenden Abschnitte. Um die Ein- und Ausgabe zu nutzen müssen Sie die Headerdatei `iostream` einbinden. Die Streamobjekte `cin` und `cout` liegen im Namensraum `std`. Auf Elemente innerhalb eines Namensraums wird mit dem Bereichsoperator `::` zugegriffen.

```
1. include <iostream>


int main(){
    std::cout << "Das wird Ausgegeben.\n";
    int wert;
    std::cin >> wert;
    std::cout << "Wert ist: " << wert << std::endl;
}
```

`endl` fügt einen Zeilenumbruch ein und leert anschließend den Ausgabepuffer.

Mit der `using`-Direktive könne auch alle Elemente eines Namensraums verfügbar gemacht werden.

```
1. include <iostream>
```

```
using namespace std;
int main(){
    cout << "Das wird Ausgegeben.\n";
    int wert;
    cin >> wert;
    cout << "Wert ist: " << wert << endl;
}
```

15.2 Kommentare

Es gibt in C++ zwei Arten von Kommentaren

```
// Kommentare über eine Zeile
/* und Kommentare über mehrere
Zeilen*/
```

15.3 Rechnen

C++ beherrscht die 4 Grundrechenarten, die Operatoren lauten +, −, * und /. Es gilt Punkt- vor Strichrechnung und Klammern ändern die Auswertungsreihenfolge. Weiterhin bietet C++ mit dem Zuweisungsoperator kombinierte Rechenoperatoren:

```
lvalue += rvalue;
lvalue -= rvalue;
lvalue *= rvalue;
lvalue /= rvalue;
```

Für Ganzzahlige Datentypen stehen weiterhin der Inkrement- und der Dekrementoperator, jeweils in einer Prä- und einer Postfixvariante zur Verfügung:

```
++lvalue; // Erhöht den Wert und gibt ihn zurück
```

```
lvalue++; // Erhöht den Wert und gibt den alten Wert zurück
-lvalue; // Erniedriegt den Wert und gibt ihn zurück
lvalue-; // Erniedriegt den Wert und gibt den alten Wert zurück
```

Die Rückgabe der Präfixvariante ist ein lvalue, die Postfixvariante gibt ein rvalue zurück.

15.4 Variablen

Die Syntax zur Deklaration einer Variable lautet:

```
«Datentyp» «Name»;
```

15.4.1 Datentypen

Die C++-Basisdatentypen sind:

- Typlos
  - `void` (wird später erläutert)

  Logisch
  - `bool`

  Zeichen (auch Ganzzahlig)
  - `char` (`signed char` und `unsigned char`)
  - `wchar_t`

  Ganzzahlig
  - `short` (`signed short` und `unsigned short`)
  - `int` (`signed int` und `unsigned int`)
  - `long` (`signed long` und `unsigned long`)

  Gleitkommazahlen
  - `float`

- `double`
- `long double`

Die Typmodifizierer `signed` und `unsigned` geben an, ob es sich um einen Vorzeichenbehafteten oder Vorzeichenlosen Typ handelt. Ohne diese Modifizierer wird ein Vorzeichenloser Typ angenommen, außer bei `char`. Hier ist es Implementierungsabhängig. `signed` und `unsigned` können ohne Datentyp verwendet werden, dann wird `int` als Typ angenommen.

Das Schlüsselwort `const` zeichnet einen Datentyp als konstant aus. Es steht links vom, zu dem es gehört, außer links steht nichts mehr, in diesem Fall gehört es zu dem was rechts davon steht. Es ist somit egal ob Sie `const` vor oder hinter einen Basisdatentyp schreiben:

```
const «Datentyp» «Variablenname»;
«Datentyp» const «Variablenname»;
```

C++ macht keine Angaben über die Größen der Datentypen, es ist lediglich folgendes festgelegt:

- `char` <= `short` <= `int` <= `long`
- `float` <= `double` <= `long double`

15.4.2 Initalisierung

In C++ gibt es 2 Möglichkeiten für eine Initialisierung:

```
Datentyp «Variablenname» = «Wert»;
Datentyp «Variablenname»(«Wert»);
```

Die erste Variante ist verbreiteter, aber nur möglich wenn zur Initialisierung nur 1 Wert benötigt wird. Alle Basisdatentypen erwarten genau einen Wert.

15.4.3 Typaufwertung

Wird ein Operator auf verschiedene Datentypen angewendet, müssen vor der Operation beide in den gleichen Typ konvertiert werden. Der Zieltyp ist üblicherweise

der kleinste, der beide Werte darstellen kann. Für genauere Informationen lesen Sie bitte im Kapitel Rechnen mit unterschiedlichen Datentypen.

15.5 Kontrollstrukturen

15.5.1 Anweisungsblöcke

Mit einem Anweisungsblock können mehrere Anweisungen zusammengefasst werden. Er kann überall stehen wo auch eine Anweisung stehen könnte. Die Syntax sieht folgendermaßen aus:

```
{
    «Anweisung»
    «...»
}
```

15.5.2 Verzweigung

Eine Verzweigung hat in C++ folgende Syntax

```
if(«Bedingung»)
    «Anweisung»
else
    «Anweisung»
```

Die Bedingung ist immer ein Logischer Wert. Zahlenwerte werden implizit in logische Werte konvertiert. `0` entspricht dabei immer `false`, alle anderen Werte werden zu `true` umgewandelt.

15.5.3 Schleifen

Es gibt 2 Arten von Schleifen, wobei die `for`-Schleife sehr mächtig ist und entsprechend öfter verwendet wird.

```
// while-Schleife (Kopfgesteuert)
while(«Bedingung»)
    «Anweisung»
// do-while-Schleife: Alle Anweisungen werden mindestens einmal ausgeführt. (Fußgesteuert)
do
    «Anweisung»
while(«Bedingung»);
// for-Schleife: Ermöglicht die Initialisierung, die Bedingung (Kopfgesteuert) und eine Aktion,
// die nach jedem Schleifendurchlauf stattfindet, im Schleifenkopf zu platzieren
for(«Deklarationen (z.B. Zählvariablen)»; «Bedingung»; «Aktion nach einem Durchlauf (z.B.
 hochzählen)»)
    «Anweisung»
```

15.5.4 Auswahl

Mit der `switch`-Anweisung, kann eine Variable auf mehrere (konstante) Werte verglichen werden. Jedem dieser Werte können eine oder mehrere Anweisungen folgen, welcher im Falle einer Übereinstimmung ausgeführt werden, bis das Ende des `switch`-Blocks erreicht wird.

```
switch(«Variable»){
    case «Wert»:
        «Anweisung»
        «...»
        »break;«
    «...»
    default: // Wird ausgeführt, falls die Variable keinem der Werte entsprach
        «Anweisung»
        «...»
}
```

Hinweis
Wird das `break;` am Ende eines Anweisungsblockes weggelassen, *werden* auch die darauf folgenden Blöcke ausgeführt, bis ein `break;` gefunden oder das Ende der `switch`-Anweisung erreicht ist.

Kapitel 16

Prozeduren und Funktionen

Unter einer Funktion (*function*, in anderen Programmiersprachen auch Prozedur oder Subroutine genannt) versteht man ein Unterprogramm, das eine bestimmte Aufgabe erfüllt. Funktionen sind unter anderem sinnvoll, um sich oft wiederholende Befehle zu kapseln, so dass diese nicht jedesmal neu geschrieben werden müssen. Zudem verbessert es die Übersichtlichkeit der Quellcode-Struktur erheblich, wenn der Programmtext logisch in Abschnitte unterteilt wird.

16.1 Parameter und Rückgabewert

Die spezielle Funktion `main()` ist uns schon mehrfach begegnet. In C++ lassen sich Funktionen nach folgenden Kriterien unterscheiden:

- Eine Funktion kann Parameter besitzen, oder nicht.
- Eine Funktion kann einen Wert zurückgeben, oder nicht.

Dem Funktionsbegriff der Mathematik entsprechen diejenigen C++-Funktionen, die sowohl Parameter haben als auch einen Wert zurückgeben. Dieser Wert kann im Programm weiter genutzt werden, um ihn z.B. einer Variablen zuzuweisen.

```
int a = f(5); // Aufruf einer Funktion
```

Damit diese Anweisung fehlerfrei kompiliert wird, muss vorher die Funktion `f()` deklariert worden sein. Bei einer Funktion bedeutet *Deklaration* die Angabe des *Funktionsprototyps*. Das heißt, der Typ von Parametern und Rückgabewert muss

angegeben werden. Das folgende Beispiel deklariert bspw. eine Funktion, die einen Parameter vom Typ `int` besitzt und einen `int`-Wert zurückgibt.

```
int f (int x); // Funktionsprototyp == Deklaration
```

Soll eine Funktion keinen Wert zurückliefern, lautet der Rückgabetyp formal `void`.

Nach dem Compilieren ist das Linken der entstanden Objektdateien zu einem ausführbaren Programm nötig. Der Linker benötigt die Definition der aufzurufenden Funktion. Eine Funktionsdefinition umfasst auch die Implementation der Funktion, d.h. den Code, der beim Aufruf der Funktion ausgeführt werden soll. In unserem Fall wäre das:

```
int f(int x);     // Funktionsdeklaration
int main(){
    int a = f(3); // Funktionsaufruf
    // a hat jetzt den Wert 9
}
int f(int x){    // Funktionsdefinition
    return x * x;
}
```

Hinweis
Innerhalb eines Programms dürfen Sie eine Funktion beliebig oft (übereinstimmend!) deklarieren, aber nur einmal definieren.

Der Compiler muss die Deklaration kennen, um eventuelle Typ-Unverträglichkeiten abzufangen. Würden Sie die obige Funktion z.B. als `int a = f(2.5);` aufrufen, käme die Warnung, dass `f()` ein ganzzahliges Argument erwartet und keine Fließkommazahl. Eine Definition ist für den Compiler auch immer eine Deklaration, das heißt Sie müssen nicht explizit eine Deklaration einfügen um eine Funktion aufzurufen, die *zuvor* definiert wurde.

Die Trennung von Deklaration und Definition kann zu übersichtlicher Code-Strukturierung bei größeren Projekten genutzt werden. Insbesondere ist es sinnvoll, Deklarationen und Definitionen in verschiedene Dateien zu schreiben. Oft will man, wenn man fremden oder alten Code benutzt, nicht die Details der Implementierung einer Funktion sehen, sondern nur das Format der Parameter o.ä. und

kann so in der Deklarationsdatei (*header file*, üblicherweise mit der Endung .hpp, z.T. auch .h oder .hh) nachsehen, ohne durch die Funktionsrümpfe abgelenkt zu werden. (Bei proprietärem Fremdcode bekommt man die Implementation in der Regel gar nicht zu Gesicht!)

Bei einer Deklaration ist es nicht nötig, die Parameternamen mit anzugeben, denn diese sind für den Aufruf der Funktion nicht relevant. Es ist allerdings üblich die Namen dennoch mit anzugeben um zu verdeutlichen was der Parameter darstellt. Der Compiler ignoriert die Namen in diesem Fall einfach, weshalb es auch möglich ist den Parametern in der Deklaration und der Definition unterschiedliche Namen zu geben. Allerdings wird davon aus Gründen der Übersichtlichkeit abgeraten.

Mit der Anweisung `return` gibt die Funktion einen Wert zurück, in unserem Beispiel `x * x`, wobei die Variable `x` als *Parameter* bezeichnet wird. Als Argument bezeichnet man eine Variable oder einen Wert, mit denen eine Funktion aufgerufen wird. Bei Funktionen mit dem Rückgabetyp `void` schreiben Sie einfach `return;` oder lassen die `return`-Anweisung ganz weg. Nach einem `return` wird die Funktion sofort verlassen, d.h. alle nachfolgenden Anweisungen des Funktionsrumpfs werden ignoriert.

Erwartet eine Funktion mehrere Argumente, so werden die Parameter durch Kommata getrennt. Eine mit

```
int g(int x, double y);
```

deklarierte Funktion könnte z.B. so aufgerufen werden:

```
int a = g(123, -4.44);
```

Für eine leere Parameterliste schreiben Sie hinter dem Funktionsnamen einfach `()`.

```
int h(){     // Deklaration von h()
  // Quellcode ...
}
int a = h(); // Aufruf von h()
```

Hinweis
Vergessen Sie beim Aufruf einer Funktion ohne Parameter nicht die leeren Klammern. Andernfalls erhalten Sie nicht den von der Funktion zurückgelieferten Wert, sondern die Adresse der Funktion. Dies ist ein „beliebter“ Anfängerfehler, daher sollten Sie im Falle eines Fehlers erst einmal überprüfen, ob Sie nicht vielleicht irgendwo eine Funktion ohne Parameter falsch aufgerufen haben.

16.2 Übergabe der Argumente

C++ kennt zwei Varianten, wie einer Funktion die Argumente übergeben werden können: *call-by-value* und *call-by-reference*.

16.2.1 call-by-value

Bei *call-by-value* (Wertübergabe) wird der *Wert* des Arguments in einen Speicherbereich kopiert, auf den die Funktion mittels Parametername zugreifen kann. Ein Werteparameter verhält sich wie eine lokale Variable, die „automatisch“ mit dem richtigen Wert initialisiert wird. Der Kopiervorgang kann bei Klassen (Thema eines späteren Kapitels) einen erheblichen Zeit- und Speicheraufwand bedeuten!

```
    1. include <iostream>


void f1(const int x) {
    x = 3 * x;       // ungültig, weil Konstanten nicht überschrieben werden dürfen
    std::cout << x << std::endl;
}
void f2(int x) {
    x = 3 * x;
    std::cout << x << std::endl;
}
int main() {
   int a = 7;
   f2(a);          // Ausgabe: 21
   f2(5);          // Ausgabe: 15
   std::cout << x; // Fehler! x ist hier nicht definiert
   std::cout << a; // a hat immer noch den Wert 7
```

```
}
```

Wird der Parameter als `const` deklariert, so darf ihn die Funktion nicht verändern (siehe erstes Beispiel). Im zweiten Beispiel kann die Variable `x` verändert werden. Die Änderungen betreffen aber nur die lokale Kopie und sind für die aufrufende Funktion nicht sichtbar.

16.2.2 call-by-reference

Die Sprache C kennt nur call-by-value. Sollen die von einer Funktion vorgenommen Änderungen auch für das Hauptprogramm sichtbar sein, müssen sogenannte *Zeiger* verwendet werden. C++ stellt ebenfalls Zeiger zur Verfügung. C++ gibt Ihnen aber auch die Möglichkeit, diese Zeiger mittels Referenzen zu umgehen. Beide sind jedoch noch Thema eines späteren Kapitels.

Im Gegensatz zu call-by-value wird bei *call-by-reference* die Speicheradresse des Arguments übergeben, also der Wert nicht kopiert. Änderungen der (Referenz-)Variable betreffen zwangsläufig auch die übergebene Variable selbst und bleiben nach dem Funktionsaufruf erhalten. Um call-by-reference anzuzeigen, wird der Operator `&` verwendet, wie Sie gleich im Beispiel sehen werden. Wird keine Änderung des Inhalts gewünscht, sollten Sie den Referenzparameter als `const` deklarieren um so den Speicherbereich vor Änderungen zu schützen. Fehler, die sich aus der ungewollten Änderung des Inhaltes einer übergebenen Referenz ergeben, sind in der Regel schwer zu finden.

Die im folgenden Beispiel definierte Funktion `swap()` vertauscht ihre beiden Argumente. Weil diese als Referenzen übergeben werden, überträgt sich das auf die Variablen mit denen die Funktion aufrufen wurde:

```
    1. include <iostream>


void swap(int &a, int &b) {
    int tmp = a; // "temporärer" Variable den Wert von Variable a zuweisen
    a = b;       // a mit dem Wert von Variable b überschreiben
    b = tmp;     // b den Wert der "temporären" Variable zuweisen (Anfangswert von Variable a)
}
int main() {
    int x = 5, y = 10;
    swap(x, y);
    std::cout << "x=" << x << " y=" << y << std::endl;
```

```
    return 0;
}
```

Ausgabe

```
x=10 y=5
```

Tip
Ob das kaufmännische Und (&) genau nach `int (int& a, ...)`, oder genau vor der Variablen `a (int &a, ...)`, oder dazwischen `(int & a, ...)` steht, ist für den Compiler nicht von Bedeutung. Auch möglich wäre `(int&a, ...)`.

Nicht-konstante Referenzen können natürlich nur dann übergeben werden, wenn das Argument tatsächlich eine Speicheradresse hat, sprich eine Variable bezeichnet. Ein Literal z.B. `123` oder gar ein Ausdruck `1 + 2` wäre hier nicht erlaubt. Bei `const`-Referenzen wäre das möglich, der Compiler würde dann eine temporäre Variable anlegen.

Thema wird Später näher erläutert...

Wir werden die oben angesprochenen Zeiger in einem späteren Kapitel ausführlich kennen lernen. Sie ermöglichen z.B. das Arbeiten mit Arrays. Soll eine Funktion mit einem Array umgehen, werden ihr die Startadresse des Arrays (ein Zeiger) und seine Größe als Parameter übergeben. Wenn das Array nur gelesen werden soll, deklariert man die Werte, auf die der Zeiger zeigt, als `const`. Das Zeichen `*` signalisiert, dass es sich um einen Zeiger handelt.

```
1. include <iostream>


void print(int const* array, int const arrayGroesse) {
    std::cout << "Array:" << std::endl;
    for (int i = 0; i < arrayGroesse; ++i) {
        std::cout << array[i] << std::endl;
    }
}
int main() {
    int array[] = { 3, 13, 113 };
    int n = sizeof(array) / sizeof(array[0]);
    print(array, n);
}
```

Ausgabe

```
Array:
3
13
113
```

16.3 Default-Parameter

Default-Parameter dienen dazu, beim Aufruf einer Funktion nicht alle Parameter explizit angeben zu müssen. Die nicht angegebenen Parameter werden mit einer Voreinstellung (*default*) belegt. Parameter, die bei einem Aufruf einer Funktion nicht angegeben werden müssen, werden auch als „fakultative Parameter“ bezeichnet.

```
int summe(int a, int b, int c = 0, int d = 0) {
    return a + b + c + d;
}
```

```
int main() {
    int x = summe(2, 3, 4, 5); //x == 14
    x = summe(2, 3, 4);        //x == 9, es wird d=0 gesetzt
    x = summe(2, 3);           //x == 5, es wird c=0, d=0 gesetzt
}
```

Standardargumente werden in der Deklaration einer Funktion angegeben, da der Compiler sie beim Aufruf der Funktion kennen muss. Im obigen Beispiel wurde die Deklaration durch die Definition gemacht, daher sind die Parameter hier in der Definition angegeben. Bei einer getrennten Schreibung von Deklaration und Definition könnte das Beispiel so aussehen:

```
int summe(int a, int b, int c = 0, int d = 0); // Deklaration
int main() {
    int x = summe(2, 3, 4, 5); //x == 14
    x = summe(2, 3, 4);        //x == 9, es wird d=0 gesetzt
    x = summe(2, 3);           //x == 5, es wird c=0, d=0 gesetzt
}
int summe(int a, int b, int c, int d) {        // Definition
    return a + b + c + d;
}
```

16.4 Funktionen überladen

Überladen (*overloading*) von Funktionen bedeutet, dass verschiedene Funktionen unter dem gleichen Namen angesprochen werden können. Damit der Compiler die Funktionen richtig zuordnen kann, müssen die Funktionen sich in ihrer Funktionssignatur unterscheiden. In C++ besteht die Signatur aus dem Funktionsnamen und ihren Parametern, der Typ des Rückgabewerts gehört nicht dazu. So ist es nicht zulässig, eine Funktion zu überladen, die den gleichen Namen und die gleiche Parameterliste wie eine bereits existierende Funktion besitzt und sich nur im Typ des Rückgabewerts unterscheidet. Das obige Beispiel lässt sich ohne Default-Parameter so formulieren:

```
int summe(int a, int b, int c, int d) {
```

```
    return a + b + c + d;
}
int summe(int a, int b, int c) {
    return a + b + c;
}
int summe(int a, int b) {
    return a + b;
}
int main() {
    // ...
}
```

16.5 Funktionen mit beliebig vielen Argumenten

Wenn die Zahl der Argumente nicht von vornherein begrenzt ist, wird als Parameterliste die sog. *Ellipse* `...` angegeben. Der Funktion werden die Argumente dann in Form einer Liste übergeben, auf die mit Hilfe der (in der Headerdatei `cstdarg` definierten) va-Makros zugegriffen werden kann.

```
1. include <cstdarg>


int summe(int a, ...) {
    int summe = 0;
    int i = a;
    va_list Parameter;              // Zeiger auf Argumentliste
    va_start(Parameter, a);         // gehe zur ersten Position der Liste
    while (i != 0){                 // Schleife, solange Zahl nicht 0 (0 ist Abbruchbedingung)
        summe += i;                 // Zahl zur Summe addieren
        i = va_arg(Parameter, int); // nächstes Argument an i zuweisen, Typ int
    }
    va_end(Parameter);              // Liste löschen
    return summe;
}
int main() {
    int x;
    x = summe(2, 3, 4, 0);          // x = 9
    x = summe(2, 3, 0);             // x = 5
    x = summe(1,1,0,1,0);           // x = 2 (da die erste 0 in der while-Schleife für Abbruch
```

```
 sorgt)
    return x;
}
```

Obwohl unspezifizierte Argumente manchmal verlockend aussehen, sollten Sie sie nach Möglichkeit vermeiden. Das hat zwei Gründe:

- Die va-Befehle können nicht erkennen, wann die Argumentliste zu Ende ist. Es muss immer mit einer expliziten Abbruchbedingung gearbeitet werden. In unserem Beispiel muss als letztes Argument eine `0` stehen, ansonsten gibt es, je nach Compiler unterschiedliche, „interessante“ Ergebnisse.
- Die va-Befehle sind *Makros*, d.h. eine strenge Typüberprüfung findet nicht statt. Fehler werden - wenn überhaupt - erst zur Laufzeit bemerkt.

16.6 Inline-Funktionen

Um den Aufruf einer Funktion zu beschleunigen, kann in die Funktionsdeklaration das Schlüsselwort `inline` eingefügt werden. Dies ist eine Empfehlung (keine Anweisung) an den Compiler, beim Aufruf dieser Funktion keine neue Schicht auf dem Stack anzulegen, sondern den Code direkt auszuführen - den Aufruf sozusagen durch den Funktionsrumpf zu ersetzen.

Da dies – wie eben schon erwähnt – nur eine Empfehlung an den Compiler ist, wird der Compiler eine Funktion nur dann tatsächlich *inline* einbauen, wenn es sich um eine kurze Funktion handelt. Ein typisches Beispiel:

```
inline int max(int a, int b) {
    return a > b ? a : b;
}
```

Thema wird Später näher erläutert...
Das Schlüsselwort `inline` wird bei der Definition angegeben. Allerdings muss der Compiler beim Aufruf den Funktionsrumpf kennen, wenn er den Code direkt einfügen soll. Für den Aufruf einer `inline`-Funktion genügt also wie immer die Deklaration. Über diese Eigenheit von `inline`-Funktionen erfahren Sie im Kapitel „Headerdateien“ mehr. Auch die Bedeutung von Deklaration und Definition wird Ihnen nach diesem Kapitel klarer sein.

Kapitel 17

Lebensdauer und Sichtbarkeit von Variablen

In diesem Kapitel werden Sie zuerst lernen, wie lange eine Variable im Speicher unseres Programmes existiert (Lebensdauer). Danach wird behandelt, an welchen Stellen wir auf eine bestimmte Variable zugreifen können (Sichtbarkeit).

17.1 Lebensdauer

17.1.1 Globale Variablen

Variablen, die auf der äußersten Ebene deklariert werden, nennt man global, weil an jeder Stelle des Programmes auf sie zugegriffen werden kann. Dabei darf die Variable nicht innerhalb einer Funktion (`main()` eingeschlossen) oder einer Klasse deklariert werden.

Diese Variablen werden meist beim Starten initialisiert und beim Beenden wieder zerstört. Damit ist klar, dass der Speicher, den sie benötigen, für die gesamte Laufzeit des Programmes belegt ist und nicht anders verwendet werden kann.

```
int var = 234; //Beispiel für eine globale Variable
int main() {
    // ...
}
```

17.1.2 Lokale Variablen

Im Gegensatz zu globalen, werden lokale Variablen in einem bestimmten Anweisungsblock (z.B. Schleifen, `if`-Abfragen oder Funktionen) deklariert. Ihre Existenz endet wenn dieser Block wieder verlassen wird.

```
void foo(int a) {
    int lok1 = 0; // lok1 ist eine Variable im Anweisungsblock von void foo(int a)
    if (a < 0) {
        int lok2 = 1; // lok2 ist eine Variable im if-Block
    } // hier wird lok2 aus dem Speicher gelöscht...
} // ...und hier lok1
```

Hinweis

Bei Schleifen ist zu beachten dass Variablen, deren Deklaration im Schleifenrumpf steht, bei jedem Durchlauf neu initialisiert werden

```
while (1) {
    int var = 0;
    std::cout << var << std::endl; // die Ausgabe ist hier immer 0
    ++var;
}
```

17.1.3 Statische Variablen

Statische Variablen (auch statische Klassenmember) werden wie globale zu Beginn des Programmes im Speicher angelegt und bei seinem Ende wieder daraus entfernt. Der Unterschied zu einer globalen Variable wird weiter unten auf dieser Seite im Teil über Sichtbarkeit geklärt.

17.1.4 Dynamisch erzeugte Variablen

Eine Variable, die mittels dem `new` Operator angefordert wird, heißt dynamisch. Sie existiert so lange bis sie durch einen Aufruf von `delete` wieder gelöscht wird.

Hinweis
Der Aufruf von `delete` ist die einzige Möglichkeit den von einer dynamisch erzeugten Variable belegten Speicher wieder frei zu geben. Geschieht dies nicht so kann es leicht zu einem Speicherleck kommen. Lesen sie bitte auch das Kapitel über den dynamischen Speicher um Informationen über `new` bzw. `delete` zu erhalten

17.1.5 Objekte und Membervariablen

Objekte werden wie normale Variablen gehandhabt, d. h. sie können global, lokal, statisch oder dynamisch erzeugt sein. Ihre Member haben die gleiche Lebensdauer wie sie selbst. Eine Ausnahme bilden statische Klassenvariablen, die von Anfang bis Ende des Programmablaufes im Speicher vorhanden sind.

17.2 Sichtbarkeit

17.2.1 Allgemein

Um Überhaupt die Chance zu haben mit einer Variablen zu arbeiten, muss diese im Quelltext bereits deklariert worden sein. Folgendes Codestück ist also falsch und würde zu einem Fehler führen.

```
// ...
var = 34; // Fehler z.B. "symbol var not found"
int var;
// ...
```

Thema wird Später näher erläutert...
Das Prinzip der Datenkapselung in der objektorientierten Programmierung finden Sie im Abschnitt über Klassen.

17.2.2 Gültigkeitsbereiche und deren Schachtelung

Jede Variable gehört zu einem bestimmten Gültigkeitsbereich (engl. *scope*). Diese legen fest wann eine Variable von uns „gesehen“ und damit benutzt werden kann. Vereinfacht gesagt bildet jedes Paar aus geschweiften Klammern (`{}`) einen eigenen Definitionsbereich.Dazu gehören beispielsweise `if`, `else`, Schleifen und Funktionen. Diese unterschiedlichen Bereiche sind nun ineinander geschachtelt ähnlich wie die berühmten Matrjoschka-Puppen mit dem Unterschied, dass die Definitionsbereiche nicht „kleiner“ werden und dass es mehrere „nebeneinander“ geben kann.

```
// das hier gehört zum globalen Bereich
int func() { // hier beginnt der Bereich der Funktion func...
    return 0;
    // ... und ist hier auch schon wieder beendet
}
int bar(int val) {  // val gehört zum Definitionsbereich "bar"
    if (val == 7) { // diese if-Anweisung hat auch ihren eigenen Gültigkeitsbereich...
       int ich_gehoer_zum_if;
    } //... der hier zu Ende ist
} // bar ende
```

17.2.3 Welche Variablen sind sichtbar?

Jetzt ist es leicht zu bestimmen mit welchen Variablen wir an einer bestimmten Stelle im Programmcode arbeiten können: Es sind diejenigen, die entweder dem derzeitigen oder einem Gültigkeitsbereich auf einer höheren Ebene angehören. Wenn wir uns erneut das Beispiel der Puppen vor Augen halten, so wird klar was hiermit gemeint ist.

Beispiel:

```
int out;
{
    int inner_1;
}
{
    int inner_2;
    //an dieser Stelle könnten wir sowohl auf out als auch auf inner_2 zugreifen, nicht jedoch
```

```
 auf inner_1;
}
```

Zusätzlich gilt noch, das von zwei Variablen gleichen Namens nur auf die weiter inner liegende zugegriffen werden kann.

```
int var=9;
{
    int var=3;
    std::cout << var << std::endl;//die Ausgabe ist 3
}
```

Kapitel 18

Schleifen mal anders – Rekursion

Jede Funktion kann sowohl andere Funktionen als auch sich selbst aufrufen. Ein solcher Selbstaufruf wird auch rekursiver Aufruf genannt. Das dahinter stehende Konzept bezeichnet man entsprechend als Rekursion.

Eine Ausnahme von dieser Regel bildet wiedereinmal die Funktion `main()`. Sie darf ausschließlich vom Betriebssystem aufgerufen werden, also weder von einer anderen Funktion, noch aus sich selbst heraus.

Eine rekursive Problemlösung ist etwas langsamer und speicheraufwendiger als eine iterative Variante (also mit Schleifen). Dafür ist der Code allerdings auch kompakter und ein „intelligenter“ Compiler ist meist in der Lage, eine Rekursion in eine Iteration umzuwandeln um somit die Nachteile aufzuheben. Sie sollten also keine Scheu haben ein Problem mit Rekursion zu lösen, wenn Sie so schneller ein richtiges Ergebnis erhalten als beim Schreiben einer iterativen Variante. Sollten dadurch im Laufe der Entwicklung eines Programms Geschwindigkeits- oder Speichernachteile auftreten, so können Sie die Funktion immer noch durch eine iterativ arbeitende ersetzen.

18.1 Fakultät

Als erstes einfaches Beispiel einer rekursiven Problemlösung nehmen wir die Berechnung der Fakultät. Da die Fakultät für negative und nicht ganze Zahlen nicht definiert ist, benutzen wir als Datentyp `unsigned int`:

```
1. include <iostream> // Für std::cin und std::cout
```

```
unsigned int fakultaet(unsigned int zahl) {
    if (zahl <= 1) {
        return 1; // Die Fakultät von 0 und 1 ist als 1 definiert.
    }
    return fakultaet(zahl - 1) * zahl;
}
int main() {
    unsigned int zahl;
    std::cout << "Bitte Zahl eingeben: ";
    std::cin >> zahl;                          // Zahl einlesen
    std::cout << "Die Fakultät von " << zahl << // Antwort ausgeben
        " ist " << fakultaet(zahl) << "!" << endl;
}
```

Ausgabe

```
Bitte Zahl eingeben: <eingabe>4</eingabe>
Die Fakultät von 4 ist 24!
```

Genau wie bei einer Schleife, ist auch bei einer Rekursion eine Abbruchbedingung definiert (also erforderlich) und genau wie bei einer Schleife würde ohne Abbruchbedingung eine Endlosrekursion auftreten, analog zur Endlosschleife. So eine Endlosschleife bezeichnet man auch als infiniten Regress. Wenn der Wert der Variablen `zahl` kleiner, oder gleich eins ist, so wird eins zurückgegeben, andernfalls wird weiter rekursiv aufgerufen. Eine iterative Variante für das gleiche Problem könnte folgendermaßen aussehen:

```
unsigned int fakultaet(unsigned int zahl) {
    unsigned int wert = 1;
    for (unsigned int i = 2; i <= zahl; ++i) {
        wert *= i;
    }
    return wert;
}
```

18.2 Fibonacci-Zahlen

Als zweites Beispiel wollen wir Fibonacci-Zahlen ausrechnen.

```
1. include <iostream>
```

```
unsigned int fibonacci(unsigned int zahl) {
    if (zahl == 0) { // Die Fibonacci-Zahl von null ist null
        return 0;
    }
    if (zahl == 1) { // Die Fibonacci-Zahl von eins ist eins
        return 1;
    }
    // Ansonsten wird die Summe der zwei vorherigen Fibonacci-Zahlen zurückgegeben
    return fibonacci(zahl - 1) + fibonacci(zahl - 2);
}
int main() {
    unsigned int zahl;
    std::cout << "Bitte Zahl eingeben: ";
    std::cin >> zahl;                                  // Zahl einlesen
    std::cout << "Die Fibonacci-Zahl von " << zahl << // Antwort ausgeben
        " ist " << fibonacci(zahl) << "!" << endl;
}
```

Ausgabe

```
Bitte Zahl eingeben: <eingabe>12</eingabe>
Die Fibonacci-Zahl von 12 ist 144!
```

Die iterative Entsprechung sieht folgendermaßen aus:

```
unsigned int fibonacci(unsigned int zahl) {
    if (zahl == 0) { // Die Fibonacci-Zahl von null ist null
        return 0;
    }
    if (zahl == 1) { // Die Fibonacci-Zahl von eins ist eins
        return 1;
    }
    unsigned int ret;
    unsigned int h1 = 0;
    unsigned int h2 = 1;
    for (unsigned int i = 1; i < zahl; ++i) {
        ret = h1 + h2; // Ergebnis ist die Summe der zwei vorhergehenden Fibonacci-Zahlen
        h1 = h2;       // und den beiden Hilfsvariablen die
        h2 = ret;      // neuen vorhergehenden Fibonacci-Zahlen
    }
    return ret;
```

```
}
```

Bei vielen komplexen Problemen eignet sich Rekursion oft besser zur Beschreibung, als eine iterative Entsprechung. Aus diesem Grund trifft man das Konzept der Rekursion in der Programmierung recht häufig an.

Kapitel 19

Zeiger

19.1 Grundlagen zu Zeigern

Zeiger (*pointer*) sind Variablen, die als Wert die Speicheradresse einer anderen Variablen enthalten.

Jede Variable wird in C++ an einer bestimmten Position im Hauptspeicher abgelegt. Diese Position nennt man *Speicheradresse* (*memory address*). C++ bietet die Möglichkeit, die Adresse jeder Variable zu ermitteln. Solange eine Variable gültig ist, bleibt sie an ein und derselben Stelle im Speicher.

Am einfachsten vergegenwärtigt man sich dieses Konzept anhand der globalen Variablen. Diese werden außerhalb aller Funktionen und Klassen deklariert und sind überall gültig. Auf sie kann man von jeder Klasse und jeder Funktion aus zugreifen. Über globale Variablen ist bereits zur Compilerzeit bekannt, wo sie sich innerhalb des Speichers befinden (also kennt das Programm ihre Adresse).

Zeiger sind nichts anderes als normale Variablen. Sie werden deklariert (und definiert), besitzen einen Gültigkeitsbereich, eine Adresse und einen Wert. Dieser Wert, der *Inhalt* der Zeigervariablen, ist aber nicht wie in unseren bisherigen Beispielen eine Zahl, sondern die Adresse einer anderen Variablen. Bei der Deklaration einer Zeigervariable wird der Typ der Variablen festgelegt, auf den sie verweisen soll. Dieser Typ ist fest und kann nicht verändert werden.

```
1. include <iostream>


int main() {
    int   Wert;      // eine int-Variable
```

```
    int *pWert;      // eine Zeigervariable, zeigt auf einen int
    int *pZahl;      // ein weiterer "Zeiger auf int"
    Wert = 10;       // Zuweisung eines Wertes an eine int-Variable
    pWert = &Wert;   // Adressoperator '&' liefert die Adresse einer Variablen
    pZahl = pWert;   // pZahl und pWert zeigen jetzt auf dieselbe Variable
```

Der *Adressoperator* `&` kann auf jede Variable angewandt werden und liefert deren Adresse, die man einer (dem Variablentyp entsprechenden) Zeigervariablen zuweisen kann. Wie im Beispiel gezeigt, können Zeiger gleichen Typs einander zugewiesen werden. Zeiger verschiedenen Typs bedürfen einer Typumwandlung. Die Zeigervariablen `pWert` und `pZahl` sind an verschiedenen Stellen im Speicher abgelegt, nur die Inhalte sind gleich.

Wollen Sie auf den Wert zugreifen, der sich hinter der im Zeiger gespeicherten Adresse verbirgt, so verwenden Sie den *Dereferenzierungsoperator* `*`.

```
   *pWert += 5;
    *pZahl += 8;
    std::cout << "Wert = " << Wert << std::endl;
}
```

Ausgabe

```
Wert = 23
```

Man nennt das *den Zeiger dereferenzieren*. Im Beispiel erhalten Sie die Ausgabe `Wert = 23`, denn `pWert` und `pZahl` verweisen ja beide auf die Variable `Wert`.

Um es noch einmal hervorzuheben: Zeiger auf Integer (`int`) sind selbst keine Integer. Den Versuch, einer Zeigervariablen eine Zahl zuzuweisen, beantwortet der Compiler mit einer Fehlermeldung oder mindestens einer Warnung. Hier gibt es nur eine Ausnahme: die Zahl 0 darf jedem beliebigen Zeiger zugewiesen werden. Ein solcher *Nullzeiger* zeigt nirgendwohin. Der Versuch, ihn zu dereferenzieren, führt zu einem Laufzeitfehler.

Thema wird Später näher erläutert...
Der Sinn von Zeigern erschließt sich vor allem Anfängern nicht unmittelbar. Das ändert sich allerdings schnell, sobald der dynamische Speicher besprochen wird.

19.2 Zeiger und `const`

Das Schlüsselwort `const` kann auf zweierlei Arten in Verbindung mit Zeigern genutzt werden:

- Um den Wert, auf den der Zeiger zeigt, konstant zu machen
- Um den Zeiger selbst konstant zu machen

Im ersteren Fall kann der Zeiger im Laufe seines Lebens auf verschiedene Objekte zeigen, diese Werte können dann allerdings (über diesen Zeiger) nicht geändert werden. Im zweiten Fall kann der Zeiger nicht auf eine andere Adresse "umgebogen" werden. Der Wert an dieser Stelle kann allerdings verändert werden. Natürlich sind auch beide Varianten in Kombination möglich.

```
int               Wert1;           // eine int-Variable
int               Wert2;           // noch eine int-Variable
int const *       p1Wert = &Wert1; // Zeiger auf konstanten int
int * const       p2Wert = &Wert1; // konstanter Zeiger auf int
int const * const p3Wert = &Wert1; // konstanter Zeiger auf konstanten int
p1Wert  = &Wert2; // geht
```

- `p1Wert = Wert2; // geht nicht, int konstant`

```
p2Wert  = &Wert2; // geht nicht, Zeiger konstant
```

- `p2Wert = Wert2; // geht`

```
p3Wert  = &Wert2; // geht nicht, int konstant
```

- `p3Wert = Wert2; // geht nicht, Zeiger konstant`

Wie Sie sich sicher noch erinnern, gehört `const` immer zu dem was links von ihm steht. Es sei denn links steht nichts mehr, dann gehört es zu dem was rechts davon steht.

19.3 Zeigerarithmetik

Zeiger sind keine Zahlen. Deshalb sind einige arithmetischen Operationen auf Zeiger nicht anwendbar und für die übrigen gelten andere Rechenregeln als in der Zahlenarithmetik. C++ kennt die Größe des Speicherbereichs, auf den ein Zeiger verweist. Inkrementieren (oder Dekrementieren) verändert die referenzierte

Adresse unter Berücksichtigung dieser Speichergröße. Das folgende Beispiel soll den Unterschied zwischen Zahlen und Zeigerarithmetik verdeutlichen:

```
    1. include <iostream>


int main() {
    std::cout << "Zahlenarithmetik" << std::endl;
    int a = 1;   // a wird 1
    std::cout << "a: " << a << std::endl;
    a++;         // a wird 2
    std::cout << "a: " << a << std::endl;
    std::cout << "Zeigerarithmetik" << std::endl;
    int *p = &a; // Adresse von a an p zuweisen
    std::cout << "p verweist auf: " << p << std::endl;
    std::cout << " Größe von int: " << sizeof(int) << std::endl;
    p++;
    std::cout << "p verweist auf: " << p << std::endl;
}
```

Ausgabe

```
Zahlenarithmetik
a: 1
a: 2
Zeigerarithmetik
p verweist auf: 0x7fff3aa60090
 Größe von int: 4
p verweist auf: 0x7fff3aa60094
```

Wie Sie sehen, erhöht sich der Wert des Zeigers nicht um eins, sondern um vier, was genau der Größe des Typs entspricht auf den er zeigt: `int`. Auf einer Platform auf der `int` eine andere Größe hat würde natürlich entsprechend dieser Größe gezählt werden. Im nächsten Beispiel sehen Sie, wie ein Zeiger auf eine weitere Zeigervariable verweist, welche ihrerseits auf einen `int`-Wert zeigt.

```
    1. include <iostream>


int main() {
    int   a  = 1;
    int  *p  = &a; // Adresse von a an p zuweisen
    int **pp = &p; // Adresse von p an pp zuweisen
    std::cout << "pp verweist auf: " << pp << std::endl;
    std::cout << " Größe von int*: " << sizeof(int*) << std::endl;
```

```
    ++pp;
    std::cout << "pp verweist auf: " << pp << std::endl;
}
```

Ausgabe

```
pp verweist auf: 0x7fff940cb6f0
 Größe von int*: 8
pp verweist auf: 0x7fff940cb6f8
```

Wie Sie sehen hat ein Zeiger auf `int` im Beispiel eine Größe von 8 Byte. Die Größe von Datentypen ist allerdings architektur-, compiler- und systembedingt. `pp` ist vom Typ „Zeiger auf Zeiger auf `int`“, was sich dann in C++ als `int**` schreibt. Um also auf die Variable "hinter" diesen beiden Zeigern zuzugreifen, muss man `**pp` schreiben.

Es spielt keine Rolle, ob man in Deklarationen `int* p;` oder `int *p;` schreibt. Einige Programmierer schreiben den Stern direkt hinter den Datentyp (`int* p`), andere schreiben ihn direkt vor Variablennamen (`int *p`) und wieder andere lassen zu beidem ein Leerzeichen (`int * p`). In diesem Buch wird die Konvention verfolgt, den Stern direkt vor Variablennamen zu schreiben wenn einer vorhanden ist (`int *p`), andernfalls wird er direkt nach dem Datentyp geschrieben (`int*`).

19.4 Negativbeispiele

Zur Verdeutlichung zwei Beispiele, die *nicht funktionieren*, weil sie vom Compiler nicht akzeptiert werden:

```
pWert = Wert;     // dem Zeiger kann kein int zugewiesen werden
Wert  = pWert;    // umgekehrt natürlich auch nicht
```

Zeigervariablen erlauben als Wert nur Adressen auf Variablen. Daher kann einer Zeigervariable wie in diesem Beispiel kein Integer-Wert zugewiesen werden.

Im Folgenden wollen wir den Wert, auf den `pWert` zeigt, inkrementieren. Einige der Beispiele bearbeiten die Adresse, die `pWert` enthält. Erst das letzte Beispiel verändert tatsächlich den Wert, auf den `pWert` zeigt. Beachten Sie, dass jede Codezeile ein Einzelbeispiel ist.

```
int Wert = 0;
int *pWert = &Wert; // pWert zeigt auf Wert
pWert += 5;         // ohne Dereferenzierung (*pWert) verändert man die Adresse, auf die
                    // der Zeiger verweist, und nicht deren Inhalt
```

```
std::cout << pWert; // es wird nicht die Zahl ausgegeben, auf die pWert
                    // zeigt, sondern deren (veränderte) Adresse
printf("Wert enthält: %d", pWert);    // gleiche Ausgabe
pWert++;            // Diese Operation verändert wiederum die Adresse, da nicht dereferenziert
 wird.
```

- `pWert++; // Auf diese Idee kommt man als nächstes. Doch auch das hat nicht den`

```
                    // gewünschten Effekt. Da der (Post-)Inkrement-Operator vor dem
 Dereferenzierungs-
                    // operator ausgewertet wird, verändert sich wieder die Adresse.
(*pWert)++;         // Da der Ausdruck in der Klammer zuerst ausgewertet wird, erreichen wir
                    // diesmal den gewünschten Effekt: Eine Änderung des Wertes.
```

19.5 `void`-Zeiger (anonyme Zeiger)

Eine besondere Rolle spielen die „Zeiger auf `void`“, die so genannten *generischen Zeiger*. Einem Zeiger vom Typ `void*` kann jeder beliebige Zeiger zugewiesen werden. `void`-Zeiger werden in der Sprache C z.B. bei der dynamischen Speicherverwaltung verwendet. In C++ kommt man weitgehend ohne sie aus. Vermeiden Sie Zeiger auf `void` wenn Sie eine andere Möglichkeit haben.

Eine Variable kann nicht vom Typ `void` sein. Daher würde folgende Zeile zu einem Fehler führen:

```
void variable;
```

Sie können einer Variable, die auf void zeigt, einen *beliebigen* Zeiger zuweisen. Deshalb werden solche Variablen meist für Zeiger verwendet, dessen Typ noch nicht feststeht und sich erst im Laufe des Programmes ergibt oder aber als temporärer Speicher mit wechselnden Zeigertypen.

1. `include <iostream>`

```
int main() {
    int intValue    = 1;
    int *intPointer = &intValue; // zeigt auf intValue
    void *voidPointer;
```

```
    voidPointer = intPointer; // voidPointer zeigt auf intPointer
```

Sie können jetzt nicht ohne Weiteres auf `*voidPointer` zugreifen, um an die Adresse von `intValue` zu bekommen. Da es sich um einen Zeiger, vom Typ `void` handelt, muss man diesen erst casten. In diesem Fall nach `int*`.

```
   std::cout << *reinterpret_cast<int*>(voidPointer) << std::endl;
}
```

Ablauf im Detail:

1. Zeiger `*voidPointer` ist vom Typ `void` und zeigt auf `intPointer`
2. `reinterpret_cast<int*>`, Zeiger ist vom Typ `int*`
3. Zeiger dereferenzieren (`*`), um Wert zu erhalten

19.6 Zeiger und Funktionen

Wenn Sie einen Zeiger als Parameter an eine Funktion übergeben, können Sie den Wert an der übergeben Adresse ändern. Eine Funktion, welche die Werte zweier Variablen vertauscht, könnte folgendermaßen implementiert werden:

```
    1. include <iostream>


void swap(int *wert1, int *wert2) {
    int tmp;
    tmp    = *wert1;
    *wert1 = *wert2;
    *wert2 = tmp;
}
int main() {
    int a = 7, b = 9;
    std::cout << "a: " << a << ", b: " << b << std::endl;
    swap(&a, &b);
    std::cout << "a: " << a << ", b: " << b << std::endl;
}
```

Ausgabe

```
a: 7, b: 9
```

```
a: 9, b: 7
```

Diese Funktion hat natürlich einige Schwachstellen. Beispielsweise stürtzt sie ab, wenn ihr ein Nullzeiger übergeben wird. Aber sie zeigt, dass es mit Zeigern möglich ist, den Wert einer Variable außerhalb der Funktion zu verändern. In Kürze werden Sie sehen, dass sich dieses Beispiel besser mit Referenzen lösen lässt.

19.7 Funktionszeiger

Zeiger können nicht nur auf Variablen, sondern auch auf Funktionen verweisen.

```
int (*pFunc)(int);   // Anlegen eines Funktionszeigers
int (Klasse::*pFunc)(int);   // Anlegen eines Funktionszeigers auf eine Memberfunktion
```

Um einen Zeiger auf eine Funktion zu deklarieren, benutzen Sie den Prototyp der Funktion und ersetzen dort den Funktionsnamen durch `(*Variablenname)`. Das ganze wird gelesen als: `pFunc` ist ein Zeiger auf eine Funktion, die einen `int` übernimmt und einen `int` zurückgibt. Ohne die Klammern um den Variablennamen müsste man lesen: `pFunc` ist eine Funktion, die einen `int` übernimmt und einen Zeiger auf `int` zurückgibt.

Dem Zeiger kann die Adresse einer typentsprechenden Funktion zugewiesen werden. Typentsprechend heißt so viel wie: Übernimmt einen `int`-Wert als Argument und gibt einen `int`-Wert zurück. Die Funktion muss also einen ihrer Zeigervariablen entsprechenden Prototyp besitzen.

```
int sqr(int x) {   // Prototyp der Funktion f
    return x * x;
}
pFunc = & sqr;   // Adresse der Funktion sqr() der Variable pFunc zuweisen
pFunc = & Klasse::sqr;   // bei Verwendung für Memberfunktionen muss hier die Klasse angegeben
 werden
```

Um die Adresse der Funktion zu erhalten, müssen Sie den Adressoperator auf den Funktionsnamen anwenden. Beachten Sie, dass die Klammern, die Sie zum Aufruf einer Funktion *immer* setzen müssen, hier keinesfalls gesetzt werden dürfen.

`&sqr()` würde Ihnen die Adresse, des von `sqr()` zurückgelieferten Objekts beschaffen. Es sei auch darauf hingewiesen, dass der Adressoperator nicht zwingend zum Ermitteln der Funktionsadresse notwendig ist. Sie sollten Ihn aus Gründen der Übersicht allerdings immer mitschreiben.

```
int x;
x = (*pFunc)(2); // ruft sqr() auf und weist x den Rückgabewert zu
x = pFunc(2);    // alternative (nicht empfohlene) Syntax
x = (*this.*pFunc)(2) // auch hier die Variante für eine Memberfunktion
```

Auch für das Aufrufen einer Funktion über einen Funktionszeiger gibt es zwei Möglichkeiten. Sie können den Zeiger erst dereferenzieren, dann benötigen Sie Klammern um die Dereferenzierung, damit nicht das zurückgegebene Objekt dereferenziert wird, oder Sie rufen die Funktion mit der gleichen Syntax auf, über die Sie dies bei einem direkten Aufruf der Funktion tun würden.

Die Syntax der zweiten Variante ist einfacher, allerdings wird dabei nicht deutlich, dass eine Funktion über einen Zeiger aufgerufen wird. Hier gehen die Meinungen darüber, welche Variante besser ist, auseinander. Fakt ist jedoch, dass die erste Variante eindeutiger das Geschehen dokumentiert.

Typisches Beispiel für den Einsatz von Funktionszeigern stellt eine Sortierroutine dar, der man die Vergleichsfunktion als Argument übergibt. Ein ausführliches (englischsprachiges) Tutorial über Funktionszeiger finden Sie unter http://www.newty.de/fpt/index.html.

19.8 Zeiger und Referenzen und Klassen

Thema wird Später näher erläutert...
Im Zusammenhang mit Klassen werden uns weitere Arten von Zeigern begegnen: • Zeiger auf statische Datenelemente • Zeiger auf Elementfunktionen. Beide werden Sie zu einem späteren Zeitpunkt noch kennenlernen.

19.9 Löschen von Zeigern

Zeiger müssen prinzipiell nicht gelöscht werden, weil diese nur Verweise auf Adressbereiche sind. Es ist dennoch möglich, mit *delete* einen Zeiger zu löschen. Dabei sollte man beachten, dass der Zeiger seinen Gültigkeitsbereich verliert. Man gibt nur den Speicher frei, auf den der Zeiger verweist. Man kann dem Zeiger aber trotzdem noch Werte zuweisen. *delete* sollte nicht aufgerufen werden, wenn ein Zeiger auf einen nicht-vorhandenen Bereich zeigt, da dies das Programm zum Absturz bringen kann. Daher sollte nach dem Aufruf von *delete*, der Wert des Zeigers auf 0 gesetzt werden, weil delete bei Null-Zeigern keinen Fehler erzeugt.

Kapitel 20

Referenzen

20.1 Grundlagen zu Referenzen

Referenzen sind Aliasnamen für Variablen. Sie werden also genau so verwendet wie gewöhnliche Variablen, enthalten jedoch das gleiche Objekt, wie die Variable mit der sie initialisiert wurden.

```
int  a = 1;  // eine Variable
int &r = a;  // Referenz auf die Variable a
std::cout << "a: " << a << " r: " << r << std::endl;
++a;
std::cout << "a: " << a << " r: " << r << std::endl;
++r;
std::cout << "a: " << a << " r: " << r << std::endl;
```

Ausgabe

```
a: 1 r: 1
a: 2 r: 2
a: 3 r: 3
```

Wie Sie im Beispiel sehen, sind `a` und `r` identisch. Gleiches können Sie natürlich auch mit einem Zeiger erreichen, auch wenn bei einem Zeiger die Syntax etwas anders ist als bei einer Referenz.

Im Beispiel wurde die Referenz auf `int r`, mit dem `int a` *initialisiert*. Beachten Sie, dass die Initialisierung einer Referenzvariablen nur beim Anlegen erfolgen kann, danach kann sie nur noch durch eine Zuweisung geändert werden. Daraus

folgt, dass eine Referenz immer initialisiert werden muss und es nicht möglich ist, eine Referenzvariable auf ein neues Objekt verweisen zu lassen:

```
int  a = 10; // eine Variable
int  b = 20; // noch eine Variable
int &r = a;  // Referenz auf die Variable a
std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
++a;
std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
r = b;       // r zeigt weiterhin auf a, r (und somit a) wird 20 zugewiesen
std::cout << "a: " << a << " b: " << b << " r: " << r << std::endl;
```

Ausgabe

```
a: 10 b: 20 r: 10
a: 11 b: 20 r: 11
a: 20 b: 20 r: 20
```

Wie Sie sehen, ist es nicht möglich, `r` als Alias für `b` zu definieren, nachdem es einmal mit `a` initialisiert wurde. Die Zuweisung bewirkt genau das, was auch eine Zuweisung von `b` an `a` bewirkt hätte. Dass eine Referenz wirklich nichts weiter ist als ein Aliasname wird umso deutlicher, wenn man sich die Adressen der Variablen aus dem ersten Beispiel ansieht:

```
int  a = 1;  // eine Variable
int &r = a;  // Referenz auf die Variable a
std::cout << "a: " << &a << " r: " << &r << std::endl;
```

Ausgabe

```
a: 0x7fffbf623c54 r: 0x7fffbf623c54
```

Wie Sie sehen, sind die Adressen identisch.

20.2 Anwendung von Referenzen

Vielleicht haben Sie sich bereits gefragt, wofür Referenzen nun eigentlich gut sind, schließlich könnte man ja auch einfach die Originalvariable benutzen. Möglicherweise erinnern Sie sich aber auch noch, dass im Kapitel „Prozeduren und Funktionen“ die Wertübergabe als Referenz (*call-by-reference*) vorgestellt wurde.

Referenzen bieten genau wie Zeiger die Möglichkeit, den Wert einer Variable außerhalb der Funktion zu ändern. Im Folgenden sehen Sie die oben vorgestellte Funktion `swap()` mit Referenzen:

```
    1. include <iostream>


void swap(int &wert1, int &wert2) {
    int tmp;
    tmp   = wert1;
    wert1 = wert2;
    wert2 = tmp;
}
int main() {
    int a = 7, b = 9;
    std::cout << "a: " << a << ", b: " << b << "\n";
    swap(a, b);
    std::cout << "a: " << a << ", b: " << b << "\n";
}
```

Ausgabe

```
a: 7, b: 9
a: 9, b: 7
```

Diese Funktion bietet gegenüber der Zeigervariante einige Vorteile. Die Syntax ist einfacher und es ist nicht möglich, so etwas wie einen Nullzeiger zu übergeben. Um diese Funktion zum Absturz zu bewegen, ist schon einige Mühe nötig.

20.2.1 `const`-Referenzen

Referenzen auf konstante Variablen spielen in C++ eine besondere Rolle. Eine Funktion die eine Variable übernimmt, kann genauso gut auch eine Referenzen auf eine konstante Variablen übernehmen. Folgendes Beispiel soll dies demonstrieren:

```
    1. include <iostream>


void ausgabe1(int wert) {
    std::cout << "wert: " << wert << "\n";
}
void ausgabe2(int const &wert) {
    std::cout << "wert: " << wert << "\n";
}
```

```
int main() {
    ausgabe1(5);
    ausgabe2(5);
}
```

Ausgabe

```
ausgabe1 wert: 5
ausgabe2 wert: 5
```

Die beiden Ausgabefunktionen sind an sich identisch, lediglich die Art der Parameterübergabe unterscheidet sich. `ausgabe1()` übernimmt einen `int`, `ausgabe2()` eine Referenz auf einen konstanten `int`. Beide Funktionen lassen sich auch vollkommen identisch aufrufen. Würde `ausgabe2()` eine Referenz auf einen nicht-konstanten `int` übernehmen, wäre ein Aufruf mit einer Konstanten, wie dem `int`-Literal `5` nicht möglich.

In Verbindung mit Klassenobjekten ist die Übergabe als Referenz auf ein konstantes Objekt sehr viel schneller, dazu erfahren Sie aber zu gegebener Zeit mehr. Für die Ihnen bereits bekannten Basisdatentypen ist tatsächlich die Übergabe als Wert effizienter.

20.2.2 Referenzen als Rückgabetyp

Referenzen haben als Rückgabewert die gleichen Vorteile wie bei der Wertübergabe. Allerdings sind Sie in diesem Zusammenhang wesentlich gefährlicher. Es kann schnell passieren, dass Sie versehentlich eine Referenz auf eine lokale Variable zurückgeben. Diese Variable ist außerhalb der Funktion allerdings nicht mehr gültig, daher ist das Resultat, wenn Sie außerhalb der Funktion darauf zugreifen, undefiniert. Aus diesem Grund sollten Sie Referenzen als Rückgabewert nur verwenden wenn Sie wirklich wissen, was Sie tun.

```
    1. include <iostream>


// gibt die Referenz des Parameters x zurück
int &zahl(int &x) {
    return x;
}
int main() {
    int y = 3;
    zahl(y) = 5;
    std::cout << y; // Ausgabe: 5
```

```
}
```

Kapitel 21

Felder

In C++ lassen sich mehrere Variablen desselben Typs zu einem *Array* (im Deutschen bisweilen auch *Feld* oder *Vektor* genannt) zusammenfassen. Auf die *Elemente* des Arrays wird über einen Index zugegriffen. Bei der Definition sind der Typ der Elemente und die Größe des Arrays anzugeben. Folgende Möglichkeiten stehen zum Anlegen eines Array zur Verfügung:

```
int feld[10];                          // Anlegen ohne Initialisierung
int feld[]   = {1,2,3,4,5,6,7,8,9,10}; // Mit Initialisierung (automatisch 10 Elemente)
int feld[10] = {1,2,3,4,5,6,7,8,9,10}; // 10 Elemente, mit Initialisierung
```

Soll das Array initialisiert werden, verwenden Sie eine Aufzählung in geschweiften Klammern, wobei der Compiler die Größe des Arrays selbst ermitteln *kann*. Es wird empfohlen, diese automatische Größenerkennung nicht zu nutzen. Wenn die Größenangabe explizit gemacht wurde, gibt der Compiler einen Fehler aus, falls die Anzahl der Intitiallisierungselemente nicht mit der Größenangabe übereinstimmt.

Wie bereits erwähnt, kann auf die einzelnen Elemente eines Arrays mit dem Indexoperator `[]` zugegriffen werden. Beim Zugriff auf Arrayelemente beginnt die Zählung bei 0. Das heißt, ein Array mit 10 Elementen enthält die Elemente 0 bis 9. Ein Arrayindex ist immer ganzzahlig.

```
    1. include <iostream>


int main() {
    int feld[10] = {1,2,3,4,5,6,7,8,9,10}; // 10 Elemente, mit Initalisierung
```

```
    for(int i = 0; i < 10; ++i) {
        std::cout << feld[i] << " ";   // Elemente 0 bis 9 ausgeben
    }
}
```

Ausgabe

```
1 2 3 4 5 6 7 8 9 10
```

Mit Arrayelementen können alle Operationen wie gewohnt ausgeführt werden.

```
feld[4] = 88;
feld[3] = 2;
feld[2] = feld[3] - 5 * feld[7 + feld[3]];
if(feld[0] < 1)
    ++feld[9];
for(int n = 0; n < 10; ++n)
    std::cout << feld[n] << std::endl;
```

Hinweis
Beachten Sie, dass der Compiler keine Indexprüfung durchführt. Wenn Sie ein nicht vorhandenes Element, z.B. `feld[297]` im Programm verwenden, kann Ihr Programm einigen Durchläufen unerwartet abstürzen. Zugriffe über Arraygrenzen erzeugen undefiniertes Verhalten. In modernen Desktop-Betriebssystemen kann das Betriebssystem einige dieser Bereichsüberschreitungen abgefangen und das Programm abbrechen („segmentation fault“). Manchmal überschreibt man auch einfach nur den eigenen Speicherbereich in anderen Variablen, was zu schwer zu findenden Bugs führt.

21.1 Zeiger und Arrays

Der Name eines Arrays wird vom Compiler (ähnlich wie bei Funktionen) als Adresse des Arrays interpretiert. In Folge dessen haben Sie neben der Möglichkeit über den Indexoperator auch die Möglichkeit, mittels Zeigerarithmetik auf die einzelnen Arrayelemente zuzugreifen.

```
1. include <iostream>


int main() {
```

```
    int feld[10] = {1,2,3,4,5,6,7,8,9,10};
    std::cout << feld[3]   << "\n"; // Indexoperator
    std::cout << *(feld + 3) << "\n"; // Zeigerarithmetik
}
```

Ausgabe

```
4
4
```

Im Normalfall werden Sie mit der ersten Syntax arbeiten, es ist jedoch nützlich zu wissen, wie Sie ein Array mittels Zeigern manipulieren können. Es ist übrigens nicht möglich, die Adresse von `feld` zu ändern. `feld` verhält sich also in vielerlei Hinsicht wie ein konstanter Zeiger auf das erste Element des Arrays. Einen deutlichen Unterschied werden Sie allerdings bemerken, wenn Sie den Operator `sizeof()` auf die Arrayvariable anwenden. Als Rückgabe bekommen Sie die Größe des gesamten Arrays. Teilen Sie diesen Wert durch die Größe eines beliebigen Arrayelements, erhalten Sie die Anzahl der Elemente im Array.

```
    1. include <iostream>


int main() {
    int feld[10];
    std::cout << "   Array Größe: " << sizeof(feld)                   << "\n";
    std::cout << " Element Größe: " << sizeof(feld[0])                << "\n";
    std::cout << "Element Anzahl: " << sizeof(feld) / sizeof(feld[0]) << "\n";
}
```

Ausgabe

```
   Array Größe: 40
 Element Größe: 4
Element Anzahl: 10
```

21.2 Mehrere Dimensionen

Auch mehrdimensionale Arrays sind möglich. Hierfür werden einfach Größenangaben für die benötigte Anzahl von Dimensionen gemacht. Das folgende Beispiel legt ein zweidimensionales Array der Größe 4×8 an, das 32 Elemente enthält. Theoretisch ist die Anzahl der Dimensionen unbegrenzt.

```
int feld[4][8];    // Ohne Initialisierung
int feld[4][8] = { // Mit Initialisierung
    {  1,  2,  3,  4,  5,  6,  7,  8},
    {  9, 10, 11, 12, 13, 14, 15, 16},
    { 17, 18, 19, 20, 21, 22, 23, 24},
    { 25, 26, 27, 28, 29, 30, 31, 32}
};
```

Wie Sie sehen, können auch mehrdimensionalen Arrays initialisiert werden. Die äußeren geschweiften Klammern beschreiben die erste Dimension mit 4 Elementen. Die inneren geschweiften Klammern beschreiben dementsprechend die zweite Dimension mit 8 Elementen. Beachten Sie, dass die inneren geschweiften Klammern lediglich der Übersicht dienen, sie sind nicht zwingend erforderlich. Dementsprechend ist es für mehrdimensionale Arrays immer nötig, die Größe aller Dimensionen anzugeben.

Genaugenommen wird eigentlich ein Array von Arrays erzeugt. In unserem Beispiel ist `feld` ein Array mit 4 Elementen vom Typ „Array mit 8 Elementen vom Typ `int`“. Dementsprechend sieht auch der Aufruf mittels Zeigerarithmetik auf einzelne Elemente aus.

```
1. include <iostream>


int main(){
    // Anlegen des Arrays mit Initialisierung von eben
    std::cout << feld[2][5]         << "\n"; // Indexoperator
    std::cout << *(*(feld + 2) + 5) << "\n"; // Zeigerarithmetik
}
```

Ausgabe

```
22
22
```

Es sind ebenso viele Dereferenzierungen wie Dimensionen nötig. Um sich vor Augen zu führen, wie die Zeigerarithmetik für diese mehrdimensionalen Arrays funktioniert, ist es nützlich, sich einfach die Adressen bei Berechnungen anzusehen:

```
1. include <iostream>


int main(){
```

```
    int feld[4][8];
    std::cout << "Größen\n";
    std::cout << "int[4][8]: " << sizeof(feld)     << "\n";
    std::cout << "   int[8]: " << sizeof(*feld)    << "\n";
    std::cout << "      int: " << sizeof(**feld)   << "\n";
    std::cout << "Adressen\n";
    std::cout << "       feld: " << feld        << "\n";
    std::cout << "   feld + 1: " << feld + 1    << "\n";
    std::cout << "(*feld) + 1: " << (*feld) + 1 << "\n";
}
```

Ausgabe

```
Größen
int[4][8]: 128
   int[8]: 32
      int: 4
Adressen
       feld: 0x7fff2be5d400
   feld + 1: 0x7fff2be5d420
(*feld) + 1: 0x7fff2be5d404
```

Wie Sie sehen, erhöht `feld + 1` die Adresse um den Wert 32 (Hexadezimal 20), was `sizeof(int[8])` entspricht. Also der Größe aller verbleibenden Dimensionen. Die erste Dereferenzierung liefert wiederum eine Adresse zurück. Wird diese um 1 erhöht, so steigt der Wert lediglich um 4 (`sizeof(int)`).

Hinweis
Beachten Sie, dass auch für mehrdimensionale Arrays keine Indexprüfung erfolgt. Greifen Sie nicht auf ein Element zu, dessen Grenzen außerhalb des Arrays liegen. Beim Array `int[12][7][9]` können Sie auf die Elemente [0..11][0..6][0..8] zugreifen. Die Zählung beginnt also auch hier immer bei 0 und endet dementsprechend 1 unterhalb der Dimensionsgröße.

21.3 Arrays und Funktionen

Arrays und Funktionen arbeiten in C++ nicht besonders gut zusammen. Sie können keine Arrays als Parameter übergeben und auch keine zurückgeben lassen. Da ein Array allerdings eine Adresse hat (und der Arrayname diese zurücklie-

fert), kann man einfach einen Zeiger übergeben. C++ bietet (ob nun zum besseren oder schlechteren) eine alternative Syntax für Zeiger bei Funktionsparametern an.

```
void funktion(int *parameter);
void funktion(int parameter[]);
void funktion(int parameter[5]);
void funktion(int parameter[76]);
```

Jeder dieser Prototypen ist gleichwertig. Die Größenangaben beim dritten und vierten den Beispiel werden vom Compiler ignoriert. Innerhalb der Funktion können Sie wie gewohnt mit dem Indexoperator auf die Elemente zugreifen. Beachten Sie, dass Sie die Arraygröße innerhalb der Funktion *nicht* mit `sizeof(arrayname)` feststellen können. Bei diesem Versuch würden Sie stattdessen die Größe eines Zeigers auf ein Array-Element erhalten.

Aufgrund dieses Verhaltens könnte man die Schreibweise des ersten Prototypen auswählen. Andere Programmierer argumentieren, dass bei der zweiten Schreibweise deutlich wird, dass der Parameter ein Array repräsentiert. Eine Größenangabe bei Arrayparametern ist manchmal anzutreffen, wenn die Funktion nur Arrays dieser Größe bearbeiten kann. Um es noch einmal zu betonen: Diese Größenangaben sind nur ein Hinweis für den Programmierer; Der Compiler wird ohne Fehler und Warnung Ihren Array mit allen Elementen übernehmen, auch wenn die beim Funktionsparameter angegebene Größe nicht mit Ihrem Array übereinstimmt.

Bei mehrdimensionalen Arrays sehen die Regeln ein wenig anders aus, da diese Arrays vom Typ *Array* sind. Wie Sie wissen ist es zulässig, Zeiger als Parameter zu übergeben. Entsprechend ist es natürlich auch möglich, einen Zeiger auf einen Array zulässig. Die folgenden Prototypen zeigen, wie die Syntax bei mehrdimensionalen Arrays aussieht.

```
void funktion(int (*parameter)[8]);
void funktion(int parameter[][8]);
void funktion(int parameter[4][8]);
```

Alle diese Prototypen haben einen Parameter vom Typ „Zeiger auf Array mit acht Elementen vom Typ `int`“. Ab der zweiten Dimension geben Sie also tatsächlich Arrays an, somit müssen Sie natürlich auch die Anzahl der Elemente zwingend angeben. Daher können Sie `sizeof()` in der Funktion verwenden, um die Größe zu ermitteln. Dies ist allerdings nicht notwendig, da Sie bereits im Vorfeld wissen, wie groß der Array ist und vom welchem Typ er ist. Die Größe berechnet sich wie

folgt: `sizeof(Typ) * Anzahl der Elemente`. In unserem Beispiel entspricht dies `4 * 8 = 32`. Auch ein zweidimensionales Array können Sie innerhalb der Funktion mit dem normalen Indexoperator zugreifen.

Beachten Sie, dass beim ersten Prototypen die Klammern zwingend notwendig sind, andernfalls hätten die eckigen Klammern auf der rechten Seite des Parameternamen Vorrang. Somit würde der Compiler dies wie oben gezeigt als einen Zeiger behandeln, natürlich unabhängig von der Anzahl der angegebenen Elemente. Ohne diese Klammern würden Sie also einen Zeiger auf einen Zeiger auf `int` deklarieren.

21.4 Lesen komplexer Datentypen

Sie kennen nun Zeiger, Referenzen und Arrays, sowie natürlich die grundlegenden Datentypen. Es kann Ihnen passieren, dass Sie auf Datentypen treffen, die all das in Kombination nutzen. Im Folgenden werden Sie lernen, solche komplexen Datentypen zu lesen und zu verstehen, wie man Sie schreibt.

Tip

Als einfache Regel zum Lesen von solchen komplexeren Datentypen können Sie sich merken:

- Es wird ausgehend vom Namen gelesen.
- Steht etwas rechts vom Namen, wird es ausgewertet.
- Steht rechts nichts mehr, wird der Teil auf der linken Seite ausgewertet.
- Mit Klammern kann die Reihenfolge geändert werden.

Die folgenden Beispiele werden zeigen, dass diese Regeln immer gelten:

```
int i;            // i ist ein int
int *j;           // j ist ein Zeiger auf int
int k[6];         // k ist ein Array von sechs Elementen des Typs int
int *l[6];        // l ist ein Array von sechs Elementen des Typs Zeiger auf int
int (*m)[6];      // m ist ein Zeiger auf ein Array von sechs Elementen des Typs int
int *(*&n)[6];    // n ist eine Referenz auf einen Zeiger auf ein Array von
                  // sechs Elementen des Typs Zeiger auf int
int *(*o[6])[5];  // o ist ein Array von sechs Elementen des Typs Zeiger auf ein
                  // Array von fünf Elementen des Typs Zeiger auf int
int **(*p[6])[5]; // p ist Array von sechs Elementen des Typs Zeiger auf ein Array
                  // von fünf Elementen des Typs Zeiger auf Zeiger auf int
```

Nehmen Sie sich die Zeit, die Beispiele nachzuvollziehen. Wenn Sie keine Probleme damit haben, sehen Sie sich das nächste sehr komplexe Beispiel an. Es soll die allgemeine Gültigkeit dieser Regel noch einmal demonstrieren:

```
int (*(*(**pFunc[5])(int*, double&))[6])();
```

`pFunc` ist ein Array mit fünf Elementen, das Zeiger auf Zeiger auf Funktionen enthält, die einen Zeiger auf `int` und eine Referenz auf `double` übernehmen und

einen Zeiger auf Arrays mit sechs Elementen vom Typ Zeiger auf Funktionen, ohne Parameter, mit einem `int` als Rückgabewert zurückgeben. Einem solchen Monstrum werden Sie beim Programmieren wahrscheinlich selten bis nie begegnen, aber falls doch, können Sie es mit den obengenannten Regeln ganz einfach entschlüsseln. Wenn Sie nicht in der Lage sind, dem Beispiel noch zu folgen, brauchen Sie sich keine Gedanken zu machen: Nur wenige Menschen sind in der Lage, sich ein solches Konstrukt überhaupt noch vorzustellen. Wenn Sie es nachvollziehen können, kommen Sie sehr wahrscheinlich mit jeder Datentypdeklaration klar.

Kapitel 22

Zeichenketten

22.1 Einleitung

In C++ gibt es keinen eingebauten Datentyp für Zeichenketten, lediglich einen für einzelne Zeichen. Da es in C noch keine Klassen gab, bediente man sich dort der einfachsten Möglichkeit, aus Zeichen Zeichenketten zu bilden: Man legte einfach einen Array von Zeichen an. C++ bietet eine komfortablere Lösung an: Die C++-Standardbibliothekenthält eine Klasse namens `string`. Um diese Klasse nutzen zu können, müssen Sie die gleichnamige Headerdatei `string` einbinden.

```
1. include <iostream>
2. include <string>


int main() {
    std::string zeichenkette;
    zeichenkette = "Hallo Welt!";
    std::cout << zeichenkette << std::endl;
}
```

Ausgabe

```
Hallo Welt!
```

Wir werden uns in diesem Kapitel mit der C++-Klasse `string` auseinandersetzen. Am Ende des Kapitels beleuchten wir den Umgang mit C-Strings (also `char`-Arrays) etwas genauer. Natürlich liegt auch `string`, wie alle Teile der Standardbibliothek, im Namensraum `std`.

22.2 Wie entsteht ein `string`-Objekt?

Zunächst sind einige Worte zur Notation von Zeichenketten in doppelten Anführungszeichen nötig. Wie Ihnen bereits bekannt ist, werden einzelne Zeichen in einfachen Anführungszeichen geschrieben. Dieser Zeichenliteral ist dann vom Typ `char`. Die doppelten Anführungszeichen erzeugen hingegen eine Instanz eines `char`-Arrays. `"Hallo Welt!"` ist zum Beispiel vom Typ `char[12]`.

Es handelt sich also um eine Kurzschreibweise, zum Erstellen von `char`-Arrays, damit Sie nicht `{'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '!', '\0'}` schreiben müssen, um eine einfache Zeichenkette zu erstellen. Was ist das `'\0'` und warum ist das Array 12 `char`s lang ist, obwohl es nur 11 Zeichen enthält? Wie bereits erwähnt, ist ein C-String ein Array von Zeichen. Da ein solcher C-String natürlich im Programmablauf Zeichenketten unterschiedlicher Längen enthalten konnte, beendete man die Zeichenkette durch ein Endzeichen: `'\0'` (Zahlenwert 0). Somit musste ein Array von Zeichen in C immer ein Zeichen länger sein, als die längste Zeichenkette, die im Programmverlauf darin gespeichert wurde.

Diese Kurzschreibweise kann aber noch mehr, als man auf den ersten Blick vermuten würde. Die eben genannte lange Notation zur Initialisierung eines Arrays funktioniert im Quelltext nur, wenn der Compiler auch weiß, von welchem Datentyp die Elemente des Arrays sein sollen. Da Zeichenliterale jedoch implizit in größere integrale Typen umgewandelt werden können, kann er den Datentyp nicht vom Typ der Elemente, die für die Initialisierung genutzt wurden ableiten:

```
1. include <string>


int main() {
    // char-Array mit 12 Elementen
    char a[] = {'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '!', '\0'};
    // int-Array mit 12 Elementen
    int  b[] = {'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '!', '\0'};
    // char-Array mit 12 Elementen
    std::string z = {'H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't', '!', '\0'};
}
```

Bei der Notation mit geschweiften Klammern ist dagegen immer bekannt, dass es sich um ein `char`-Array handelt. Entsprechend ist die Initialisierung eines `int`-Arrays damit nicht möglich. Folgendes dagegen schon:

```
1. include <string>
```

```
int main() {
    // char-Array mit 12 Elementen
    char a[] = "Hallo Welt!";
    // char-Array mit 12 Elementen
    std::string z = "Hallo Welt!";
}
```

Bei der Erzeugung eines `string`-Objekts wird eine Funktion aufgerufen, die sich Konstruktor nennt. Was genau ein Konstruktor ist, erfahren Sie im Kapitel über Klassen. In unserem Fall wird also der Konstruktor für das `string`-Objekt `z` aufgerufen. Als Parameter erhält er das `char`-Array `"Hallo Welt!"`. Wie Ihnen bereits bekannt ist, können an Funktionen keine Arrays übergeben werden. Stattdessen wird natürlich ein Zeiger vom Arrayelementtyp (also `char`) übergeben. Dabei geht aber die Information verloren, wie viele Elemente dieses Array enthält und an dieser Stelle kommt das `'\0'`-Zeichen (Nullzeichen) ins Spiel. Anhand dieses Zeichens kann auch innerhalb der Zeichenkette erkannt werden, wie lang die übergebene Zeichenkette ist.

Damit wissen Sie nun, wie aus dem einfachen `char`-Array das fertige `string`-Objekt wird. Jetzt ist es an der Zeit zu erfahren, was Sie mit diesem Objekt alles machen können

22.3 `string` und andere Datentypen

Wie Sie bereits im Beispiel von eben gesehen haben, lässt sich die `string`-Klasse problemlos mit anderen Datentypen und Klassen kombinieren. Im ersten Beispiel dieses Kapitels wurde zunächst eine Zuweisung eines `char`-Arrays vorgenommen. Anschließend wurde das `string`-Objekt über `cout` ausgegeben. Auch die Eingabe einer Zeichenkette über `cin` ist mit einem `string`-Objekt problemlos möglich:

```
1. include <iostream>
2. include <string>


int main() {
    std::string zeichenkette;
    std::cin >> zeichenkette;
    std::cout << zeichenkette;
```

```
}
```

Diese Art der Eingabe erlaubt es lediglich, bis zum nächsten Whitespace einzulesen. Es kommt jedoch häufig vor, dass man eine Zeichenkette bis zum Zeilenende oder einem bestimmten Endzeichen einlesen möchte. In diesem Fall ist die Funktion `getline` hilfreich. Sie erwartet als ersten Parameter einen Eingabestream und als zweiten ein `string`-Objekt.

```
1. include <iostream>
2. include <string>


int main() {
    std::string zeichenkette;
    // Liest bis zum Zeilenende
    std::getline(std::cin, zeichenkette);
    std::cout << zeichenkette;
}
```

Als optionalen dritten Parameter kann man das Zeichen angeben, bis zu dem man einlesen möchte. Im Fall von eben wurde als der Default-Parameter `'\n'` (Newline-Zeichen) benutzt. Im folgenden Beispiel wird stattdessen bis zum ersten kleinen y eingelesen.

```
1. include <iostream>
2. include <string>


int main() {
    std::string zeichenkette;
    // Liest bis zum nächsten y
    std::getline(std::cin, zeichenkette, 'y');
    std::cout << zeichenkette;
}
```

22.4 Zuweisen und Verketten

Genau wie die Basisdatentypen, lassen sich auch `string`s einander zuweisen. Für die Verkettung von `string`s wird der +-Operator benutzt und das Anhängen einer Zeichenkette ist mit += möglich.

```
1. include <iostream>
2. include <string>


int main() {
    std::string string1, string2, string3;
    string1 = "ich bin ";
    string2 = "doof";
    string3 = string1 + string2;
    std::cout << string3 << std::endl;
    string3 += " - " + string1 + "schön";
    std::cout << string3 << std::endl;
    std::cout << string1 + "schön " + string2 << std::endl;
}
```

Ausgabe

```
ich bin doof
ich bin doof - ich bin schön
ich bin schön doof
```

Spielen Sie einfach ein wenig mit den Operatoren, um den Umgang mit ihnen zu lernen.

22.5 Nützliche Methoden

Die `string`-Klasse stellt einige nützliche Methoden bereit. Etwa um den String mit etwas zu füllen, ihn zu leeren oder über verschiedene Eigenschaften Auskunft zu bekommen. Eine Methode wird mit folgender Syntax aufgerufen:

```
«stringname».«methodenname»(«parameter...»);
```

Die Methoden `size()` und `length()` erwarten keine Parameter und geben beide die aktuelle Länge der gespeicherten Zeichenkette zurück. Diese Doppelung in der

Funktionalität existiert, da `string` in Analogie zu den anderen Containerklassen der C++-Standardbibliothek `size()` anbieten muss, der Name `length()` für die Bestimmung der Länge eines Strings aber natürlicher und auch allgemein üblich ist. `empty()` gibt `true` zurück falls der String leer ist, andernfalls `false`.

Mit `clear()` lässt sich der String leeren. Die `resize()`-Methode erwartet ein oder zwei Parameter. Der erste ist die neue Größe des Strings, der zweite das Zeichen, mit dem der String aufgefüllt wird, falls die angegebene Länge größer ist, als die aktuelle. Wird der zweite Parameter nicht angegeben, wird der String mit `'\0'` (Nullzeichen) aufgefüllt. In der Regel werden Sie dieses Verhalten nicht wollen, geben Sie also ein Füllzeichen an, falls Sie sich nicht sicher sind, was Sie tun. Ist die angegebene Länge geringer, als die des aktuellen Strings, wird am Ende abgeschnitten.

Um den Inhalt zweier Strings auszutauschen existiert die `swap()`-Methode. Sie erwartet als Parameter den String mit dem ihr Inhalt getauscht werden soll. Dies ist effizienter, als das Vertauschen über eine dritte, temporäre string-Variable.

```
1. include <iostream>
2. include <string>


int main() {
    std::string zeichenkette1 = "Ich bin ganz lang!";
    std::string zeichenkette2 = "Ich kurz!";
    std::cout << zeichenkette1 << std::endl;
    std::cout << zeichenkette2 << std::endl;
    zeichenkette1.swap(zeichenkette2);
    std::cout << zeichenkette1 << std::endl;
    std::cout << zeichenkette2 << std::endl;
}
```

Ausgabe

```
Ich bin ganz lang!
Ich kurz!
Ich kurz!
Ich bin ganz lang!
```

22.6 Zeichenzugriff

Genau wie bei einem Array können Sie den `[]`-Operator (Zugriffsoperator) verwenden, um auf einzelne Zeichen im String zuzugreifen. Allerdings wird, ebenfalls genau wie beim Array, nicht überprüft, ob der angegebene Wert noch innerhalb der enthaltenen Zeichenkette liegen.

Alternativ existiert die Methode `at()`, die den Index als Parameter erwartet und eine Grenzprüfung ausführt. Im Fehlerfall löst sie eine `out_of_range`-Exception aus. Da Sie den Umgang mit Exceptions wahrscheinlich noch nicht beherrschen, sollten Sie diese Methode vorerst nicht einsetzten und stattdessen genau darauf achten, dass Sie nicht versehentlich über die Stringlänge hinaus zugreifen.

```
1. include <iostream>
2. include <string>


int main() {
    std::string zeichenkette = "Ich bin ganz lang!";
    std::cout << zeichenkette[4] << std::endl;
    std::cout << zeichenkette.at(4) << std::endl;
    std::cout << zeichenkette[20] << std::endl;    // Ausgabe von Datenmüll
    std::cout << zeichenkette.at(20) << std::endl; // Laufzeitfehler
}
```

Ausgabe

```
b
b
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::at
Abgebrochen
```

Hinweis
Beachten Sie beim Zugriff, dass das erste Zeichen den Index 0 hat. Das letzte Zeichen hat demzufolge den Index `zeichenkette.length() - 1`.

22.7 Manipulation

22.7.1 Suchen

Die Methode `find()` sucht das erste Vorkommen eines Strings und gibt die Startposition (Index) zurück. Der zweite Parameter gibt an, ab welcher Position des Strings gesucht werden soll.

```
1. include <iostream>
2. include <string>


int main() {
    std::string str = "Zeichenkette";
    std::string find = "k";
    std::cout << str.find(find, 0);
}
```

Ausgabe

```
7
```

Wird ein Substring nicht gefunden, gibt `find()` den Wert `std::string::npos` zurück.

Das Gegenstück zu `find()` ist `rfind()`. Es ermittelt das letzte Vorkommen eines Strings. Die Parameter sind die gleichen wie bei `find()`.

22.7.2 Ersetzen

Sie können `replace()` verwenden, um Strings zu ersetzen. Dafür benötigen Sie die Anfangsposition und die Anzahl der Zeichen, die anschließend ersetzt werden sollen.

```
1. include <iostream>
2. include <string>


int main() {
    std::string str = "Zeichenkette";
    str.replace(str.find("k"), std::string("kette").length(), "test");
    std::cout << str << std::endl;
}
```

Ausgabe

```
Zeichentest
```

Wie Sie sehen, verwenden wir `find()`, um die Startposition zu ermitteln. Der zweite Parameter gibt die Länge an. Hier soll die alternative Schreibweise verdeutlicht werden; Sie müssen nicht eine zusätzliche Variable deklarieren, sondern können die `std::string`-Klasse wie eine Funktion verwenden und über Rückgabewert auf die Methode `length()` zugreifen. Im dritten Parameter spezifizieren Sie den String, welcher den ursprünglichen String zwischen der angegebenen `Startposition` und `Startposition + Länge` ersetzt.

22.7.3 Einfügen

Die Methode `insert()` erlaubt es Ihnen, einen String an einer bestimmten Stelle einzufügen.

```
1. include <iostream>
2. include <string>


int main() {
    std::string str = "Hallo Welt.";
    str.insert(5, " schöne");
    std::cout << str << std::endl;
}
```

Ausgabe

```
Hallo schöne Welt
```

22.7.4 Kopieren

Mit der Methode `substr()` kann man sich einen Substring zurückgeben lassen. Der erste Parameter gibt den Startwert an. Zusätzlich kann man mit dem zweiten Parameter noch die Endposition festlegen. Wird die Methode nur mit dem Startwert aufgerufen, gibt sie alle Zeichen ab dieser Position zurück.

```
1. include <iostream>
2. include <string>


int main() {
    std::string str = "Hallo Welt.";
```

```
    std::cout << str.substr(0, str.find(' ')) << std::endl;
}
</source>
```

Ausgabe

```
Hallo
```

22.8 Vergleiche

C++-Strings können Sie, genau wie Zahlen, miteinander vergleichen. Was Gleichheit und Ungleichheit bei einem String bedeutet, wird Ihnen sofort klar sein. Sind alle Zeichen zweier Strings identisch, so sind beide gleich, andernfalls nicht. Die Operatoren <, >>, <= und >= geben da schon einige Rätsel mehr auf.

Im Grunde kennen Sie die Antwort bereits. Zeichen sind in C++ eigentlich Zahlen. Sie werden zu Zeichen, indem den Zahlen entsprechende Symbole zugeordnet werden. Der Vergleich erfolgt also einfach mit den Zahlen, welche die Zeichen kodieren. Das erste Zeichen der Strings, das sich unterscheidet, entscheidet darüber, welcher der Strings größer bzw. kleiner ist.

Die meisten Zeichenkodierungen beinhalten in den ersten 7 Bit den ASCII-Code, welchen die nachfolgende Tabelle zeigt.

ASCII-Codetabelle, Nummerierung im Hexadezimalsystem (Teil 1)								
Code	...0	...1	...2	...3	...4	...5	...6	...7
0...	*NUL*	*SOH*	*STX*	*ETX*	*EOT*	*ENQ*	*ACK*	*BEL*
1...	*DLE*	*DC1*	*DC2*	*DC3*	*DC4*	*NAK*	*SYN*	*ETB*
2...	*SP*	!	"	#	$	%	&	’
3...	0	1	2	3	4	5	6	7
4...	@	A	B	C	D	E	F	G
5...	P	Q	R	S	T	U	V	W
6...	‘	a	b	c	d	e	f	g
7...	p	q	r	s	t	u	v	w

ASCII-Codetabelle, Nummerierung im Hexadezimalsystem (Teil 2)								
Code	...8	...9	...A	...B	...C	...D	...E	...F
0...	*BS*	*HT*	*LF*	*VT*	*FF*	*CR*	*SO*	*SI*
1...	*CAN*	*EM*	*SUB*	*ESC*	*FS*	*GS*	*RS*	*US*
2...	(	)	*	+	,	-	.	/
3...	8	9	:	;	<	=	>	?
4...	H	I	J	K	L	M	N	O
5...	X	Y	Z	[	\	]	^	_
6...	h	i	j	k	l	m	n	o
7...	x	y	z	{	|	}	~	*DEL*

```
1. include <string>


int main(){
    std::string gross = "Ich bin ganz groß!";
    std::string klein = "Ich bin ganz klein!";
    gross == klein; // ergibt false ('g' != 'k')
    gross != klein; // ergibt true  ('g' != 'k')
    gross <  klein; // ergibt true  ('g' <  'k')
    gross >  klein; // ergibt false ('g' <  'k')
    gross <= klein; // ergibt true  ('g' <  'k')
    gross >= klein; // ergibt false ('g' <  'k')
}
```

22.9 Zahl zu `string` und umgekehrt

In C++ gibt es, im Gegensatz zu vielen anderen Programmiersprachen, keine Funktion, um direkt Zahlen in Strings oder umgekehrt umzuwandeln. Es ist allerdings nicht besonders schwierig, eine solche Funktion zu schreiben. Wir haben für die Umwandlung zwei Möglichkeiten:

- Die C-Funktionen `atof()`, `atoi()`, `atol()` und `sprintf()`
- C++-String-Streams

Die C-Variante wird in Kürze im Zusammenhang mit C-Strings besprochen. Für den Moment wollen wir uns der C++-Variante widmen. Stringstreams funktionieren im Grunde genau wie die Ihnen bereits bekannten Ein-/Ausgabestreams `cin` und `cout` mit dem Unterschied, dass sie ein `string`-Objekt als Ziel benutzen.

```
    1. include <iostream> // Standard-Ein-/Ausgabe
    2. include <sstream> // String-Ein-/Ausgabe


int main() {
    std::ostringstream strout; // Unser Ausgabe-Stream
    std::string str;           // Ein String-Objekt
    int var = 10;              // Eine ganzzahlige Variable
    strout << var;             // ganzzahlige Variable auf Ausgabe-Stream ausgeben
    str = strout.str();        // Streaminhalt an String-Variable zuweisen
    std::cout << str << std::endl; // String ausgeben
}
```

Der vorliegende Code wandelt eine Ganzzahl in einen String um, indem die Ganzzahl auf dem Ausgabe-Stringstream ausgegeben und dann der Inhalt des Streams an den String zugewiesen wird. Die umgekehrte Umwandlung funktioniert ähnlich. Natürlich verwenden wir hierfür einen Eingabe-Stringstream (`istringstream` statt `ostringstream`) und übergeben den Inhalt des Strings an den Stream, bevor wir ihn von diesem auslesen.

```
    1. include <iostream> // Standard-Ein-/Ausgabe
    2. include <sstream> // String-Ein-/Ausgabe


int main() {
    std::istringstream strin; // Unser Eingabe-Stream
    std::string str = "17";   // Ein String-Objekt
    int var;                  // Eine ganzzahlige Variable
    strin.str(str);           // Streaminhalt mit String-Variable füllen
    strin >> var;             // ganzzahlige Variable von Eingabe-Stream einlesen
    std::cout << var << std::endl; // Zahl ausgeben
}
```

Statt `istringstream` und `ostringstream` können Sie übrigens auch ein `stringstream`-Objekt verwenden, welches sowohl Ein-, als auch Ausgabe erlaubt, allerdings sollte man immer so präzise wie möglich angeben, was der Code machen soll. Daher ist die Verwendung eines spezialisierten Streams zu empfehlen, wenn Sie nur die speziellen Fähigkeiten (Ein- oder Ausgabe) benötigen.

Sicher sind Sie jetzt bereits in der Lage, zwei Funktionen zu schreiben, welche diese Umwandlung durchführt. Allerdings stehen wir in dem Moment, wo wir andere Datentypen als `int` in Strings umwandeln wollen vor einem Problem. Wir

können der folgenden Funktion zwar ohne weiteres eine `double`-Variable übergeben, allerdings wird dann der Nachkommateil einfach abgeschnitten. Als Lösung kommt Ihnen nur eventuell in den Sinn, einfach eine `double`-Variable von der Funktion übernehmen zu lassen.

```
1. include <iostream> // Standard-Ein-/Ausgabe
2. include <sstream> // String-Ein-/Ausgabe


std::string zahlZuString(double wert) {
    std::ostringstream strout; // Unser Ausgabe-Stream
    std::string str;           // Ein String-Objekt
    strout << wert;            // Zahl auf Ausgabe-Stream ausgeben
    str = strout.str();        // Streaminhalt an String-Variable zuweisen
    return str;                // String zurückgeben
}
int main() {
    std::string str;
    int    ganzzahl  = 19;
    double kommazahl = 5.55;
    str = zahlZuString(ganzzahl);
    std::cout << str << std::endl;
    str = zahlZuString(kommazahl);
    std::cout << str << std::endl;
}
```

Ausgabe

```
19
5.55
```

Nun, so weit so gut. Das funktioniert. Leider gibt es da aber auch noch die umgekehrte Umwandlung und obgleich es möglich ist, sie auf ähnliche Weise zu lösen, wird Ihr Compiler sich dann ständig mit einer Warnung beschweren, wenn das Ergebnis ihrer Umwandlung an eine ganzzahlige Variable zugewiesen wird.

Besser wäre es, eine ganze Reihe von Funktionen zu erzeugen, von denen jede für einen Zahlentyp verantwortlich ist. Tatsächlich können Sie in C++ mehrere Funktionen gleichen Namens erzeugen, die unterschiedliche Parameter(typen) übernehmen. Diese Vorgang nennt sich *Überladen* von Funktionen. Der Compiler entscheidet dann beim Aufruf der Funktion anhand der übergeben Parameter, welche Version gemeint war während der Programmierer immer den gleichen Namen verwendet.

Im Moment haben wir obendrein einen Sonderfall der Überladung. Alle unsere Funktionen besitzen exakt den gleichen Code. Lediglich der Parametertyp ist unterschiedlich. Es wäre ziemlich zeitaufwendig und umständlich, den Code immer wieder zu kopieren, um dann nur den Datentyp in der Parameterliste zu ändern. Noch schlimmer wird es, wenn wir eines Tages eine Änderung am Funktionsinhalt vornehmen und diese dann auf alle Kopien übertragen müssen.

Glücklicherweise bietet C++ für solche Fälle so genannte Templates, die es uns erlauben, den Datentyp vom Compiler ermitteln zu lassen. Wir teilen dem Compiler also mit, was er tun soll, womit muss er dann selbst herausfinden. Die Funktion `zahlZuString()` (umbenannt in `toString()`) sieht als Template folgendermaßen aus:

```
1. include <iostream> // Standard-Ein-/Ausgabe
2. include <sstream> // String-Ein-/Ausgabe


template <typename Typ>
std::string toString(Typ wert) {
    std::ostringstream strout; // Unser Ausgabe-Stream
    std::string str;           // Ein String-Objekt
    strout << wert;            // Zahl auf Ausgabe-Stream ausgeben
    str = strout.str();        // Streaminhalt an String-Variable zuweisen
    return str;                // String zurückgeben
}
int main() {
    std::string str;
    int    ganzzahl  = 19;
    double kommazahl = 5.55;
    std::string nochnString = "Blödsinn";
    str = toString(ganzzahl);
    std::cout << str << std::endl;
    str = toString(kommazahl);
    std::cout << str << std::endl;
    str = toString(nochnString);
    std::cout << str << std::endl;
}
```

Ausgabe

```
19
5.55
Blödsinn
```

Die letzte Ausgabe zeigt deutlich warum die Funktion in `toString` umbenannt wurde, denn sie ist nun in der Lage, jeden Datentyp, der sich auf einem `ostringstream` ausgeben lässt, zu verarbeiten und dazu zählen eben auch `string`-Objekte und nicht nur Zahlen. Sie werden später noch lernen, welches enorme Potenzial diese Technik in Zusammenhang mit eigenen Datentypen hat. An dieser Stelle sei Ihnen noch die Funktion zur Umwandlung von Strings in Zahlen (oder besser: alles was sich von einem `istringstream` einlesen lässt) mit auf den Weg gegeben:

```
1. include <iostream> // Standard-Ein-/Ausgabe
2. include <sstream> // String-Ein-/Ausgabe


template <typename Typ>
void stringTo(std::string str, Typ &wert) {
    std::istringstream strin; // Unser Eingabe-Stream
    strin.str(str);           // Streaminhalt mit String-Variable füllen
    strin >> wert;            // Variable von Eingabe-Stream einlesen
}
int main() {
    std::string str = "7.65Blödsinn";
    int    ganzzahl;
    double kommazahl;
    std::string nochnString;
    stringTo(str, ganzzahl);
    std::cout << ganzzahl << std::endl;
    stringTo(str, kommazahl);
    std::cout << kommazahl << std::endl;
    stringTo(str, nochnString);
    std::cout << nochnString << std::endl;
}
```

Ausgabe

```
7
7.65
7.65Blödsinn
```

Die Variable, die mit dem Wert des Strings belegt werden soll, wird als Referenz an die Funktion übergeben, damit der Compiler ihren Typ feststellen kann. Die Ausgabe zeigt, dass immer nur so viel eingelesen wird, wie der jeweilige Datentyp (zweiter Funktionsparameter) fassen kann. Für eine ganzzahlige Variable wird nur

die Zahl *Sieben* eingelesen, die Gleitkommavariable erhält den Wert `7.65` und das `string`-Objekt kann die gesamte Zeichenkette übernehmen.

Thema wird Später näher erläutert...
Sie werden Überladung im Kapitel „Überladen...“ (Abschnitt „Eigene Datentypen definieren“) genauer kennen lernen. Templates sind ein sehr umfangreiches Thema, Sie werden ihnen im Abschnitt Templates wiederbegegnen.

22.10 C-Strings

Wie bereits erwähnt, handelt es sich bei einem C-String um ein Array von `chars`. Das Ende eines C-Strings wird durch ein Nullzeichen (Escape-Sequenz `'\0'`) angegeben. Das Arbeiten mit C-Strings ist mühsam, denn es muss immer sichergestellt sein, dass das Array auch groß genug ist, um den String zu beinhalten. Da in C/C++ jedoch auch keine Bereichsüberprüfung durchgeführt wird, macht sich ein Pufferüberlauf (also eine Zeichenkette die größer ist als das Array, das sie beinhaltet) erst durch einen eventuellen Programmabsturz bemerkbar. Allein um dies zu vermeiden sollten Sie, wann immer es Ihnen möglich ist, die C++-string-Klasse verwenden.

Ein weiteres Problem beim Umgang mit C-Strings ist der geringe Komfort beim Arbeiten. Ob Sie einen String mit einem anderen vergleichen wollen, oder ihn an ein anderes Array „zuweisen“ möchten, in jedem Fall benötigen Sie unintuitive Zusatzfunktionen. Diese Funktionen finden Sie in der Standardheaderdatei „cstring“. Wie diese Funktionen heißen und wie man mit ihnen umgeht können Sie im C++-Referenz-Buch nachlesen, falls Sie sie einmal benötigen sollten.

Buchempfehlung
Wenn Sie sich eingehender mit der Thematik auseinandersetzen möchten, sei Ihnen das Buch C-Programmierung ans Herz gelegt. Wenn Sie in C++ mit der C-Standard-Bibliotheken arbeiten möchten, müssen Sie den Headerdateien ein „c“ voranstellen und das „.h“ weglassen. So wird beispielsweise aus dem C-Header „string.h“ der C++-Header „cstring“.

Kapitel 23

Vorarbeiter des Compilers

Bevor der Compiler eine C++-Datei zu sehen kriegt, läuft noch der Präprozessor durch. Er überarbeitet den Quellcode, sodass der Compiler daraus eine Objektdatei erstellen kann. Diese werden dann wiederum vom Linker zu einem Programm gebunden. In diesem Kapitel soll es um den Präprozessor gehen, wenn Sie allgemeine Informationen über Präprozessor, Compiler und Linker brauchen, dann lesen Sie das Kapitel „Compiler“.

23.1 `#include`

Die Präprozessordirektive `#include` haben Sie schon häufig benutzt. Sie fügt den Inhalt der angegebenen Datei ein. Dies ist nötig, da der Compiler immer nur eine Datei übersetzen kann. Viele Funktionen werden aber in verschiedenen Dateien benutzt. Daher definiert man die Prototypen der Funktionen (und einige andere Dinge, die Sie noch kennenlernen werden) in so genannten Headerdateien. Diese Headerdateien werden dann über `#include` eingebunden, weshalb Sie die Funktionen usw. Aufrufen können.

Thema wird Später näher erläutert...
Ausführlich Informationen über Headerdateien erhalten Sie im gleichnamigen Kapitel.

`#include` bietet zwei Möglichkeiten Headerdateien einzubinden:

```
1. include "name" // Sucht im aktuellen Verzeichnis und dann in den Standardpfaden des
   Compilers

2. include <name> // Sucht gleich in den Standardpfaden des Compilers
```

„Aktuelles Verzeichnis“ bezieht sich immer auf das Verzeichnis, in welchem die Datei liegt.

Die erste Syntax funktioniert immer, hat aber den Nachteil, dass dem Compiler nicht mitgeteilt wird, dass es sich um eine Standardheaderdatei handelt. Wenn sich im aktuellen Verzeichnis beispielsweise eine Datei namens `iostream` befindet und Sie versuchen über die erste Syntax, die Standardheaderdatei `iostream` einzubinden, werden Sie stattdessen die Datei im aktuellem Verzeichnis einbinden.

Aus diesem Grund hat es sich eingebürgert, wenn es möglich ist die Syntax mit den spitzen Klammern zu verwenden. Nur, wenn es nötig ist, werden die Anführungszeichen benutzt.

23.2 `#define` und `#undef`

`#define` belegt eine Textsubstitution mit dem angegebenen Wert, z.B.:

```
1. define BEGRUESSUNG "Hallo Welt!\n"


cout << BEGRUESSUNG;
```

Makros sind auch möglich, z.B.:

```
1. define ADD_DREI(x,y,z) x+y+z+3


int d(1),e(20),f(4),p(0);
p = ADD_DREI(d,e,f);              // p = d+e+f+3;
```

Diese sind allerdings mit Vorsicht zu genießen, da durch die strikte Textsubstitution des Präprozessors ohne jegliche Logikprüfungen des Compilers, fatale Fehler einfließen können und sollten eher durch (`inline`)-Funktionen, Konstanten oder sonstige Konzepte realisiert werden. Manchmal lässt es sich allerdings nicht umgehen, ein Makro (weiter) zu verwenden. Beachten Sie bitte immer, dass es sich hierbei um Anweisungen an eine Rohquelltextvorbearbeitungsstufe handelt und nicht um Programmcode.

Da anstelle von einfachen Werten auch komplexere Terme als Parameter für das Makro verwendet werden können und das Makro selbst auch in einen Term eingebettet werden kann, sollten immer Klammern verwendet werden. Bsp.:

```
1. define MUL_NOK(x,y) x*y
2. define MUL_OK(x,y) ((x)*(y))


resultNok = a * MUL_NOK(b,c+d); // a * b*c+d      = a*b*c+d        <= falsch
resultOk = a * MUL_OK(b,c+d);   // a * ((b)*(c+d)) = a*b*c + a*b*d  <= richtig
```

`#undef` löscht die Belegung einer Textsubstitution/Makro, z.B.:

```
1. undef BEGRUESSUNG
```

23.3

Der #-Ausdruck erlaubt es, den einem Makro übergebenen Parameter als Zeichenkette zu interpretieren:

```
1. define STRING(s) #s


cout << STRING(Test) << endl;
```

Das obige Beispiel gibt also den Text "Test" auf die Standard-Ausgabe aus.

23.4

Der ##-Ausdruck erlaubt es, in Makros definierte Strings innerhalb des Präprozessorlaufs miteinander zu kombinieren, z.B.:

```
1. define A "Hallo"
2. define B "Welt"
3. define A_UND_B A##B
```

Als Resultat beinhaltet die Konstante A_UND_B die Zeichenkette "HalloWelt". Der ##-Ausdruck verbindet die Namen der symbolischen Konstanten, nicht deren Werte. Ein weiteres Anwendungsbeispiel:

```
    1. define MAKE_CLASS( NAME ) \


class NAME                      \
{                               \
  public:                       \
    static void Init##NAME() {}; \
};
...
MAKE_CLASS( MyClass )
...
MyClass::InitMyClass();
```

23.5 #if, #ifdef, #ifndef, #else, #elif und #endif

Direktiven zur *bedingten Übersetzung*, d.h. Programmteile werden entweder übersetzt oder ignoriert.

```
    1. if Ausdruck1

// Programmteil 1
    1. elif Ausdruck2

// Programmteil 2
/*
   ...
 */
    1. else

// Programmteil sonst
    1. endif
```

Die Ausdrücke hinter #if bzw. #elif werden der Reihe nach bewertet, bis einer von ihnen einen von 0 verschiedenen Wert liefert. Dann wird der zugehörige

Programmteil wie üblich verarbeitet, die restlichen werden ignoriert. Ergeben alle Ausdrücke 0, wird der Programmteil nach `#else` verarbeitet, sofern vorhanden.

Als Bedingungen sind nur *konstante Ausdrücke* erlaubt, d.h. solche, die der Präprozessor tatsächlich auswerten kann. Definierte Makros werden dabei expandiert, verbleibende Namen durch 0L ersetzt. Insbesondere dürfen keine Zuweisungen, Funktionsaufrufe, Inkrement- und Dekrementoperatoren vorkommen. Das spezielle Konstrukt

```
1. defined Name
```

wird durch 1L bzw. 0L ersetzt, je nachdem, ob das Makro *Name* definiert ist oder nicht.

`#ifdef` ist eine Abkürzung für `#if defined`.

`#ifndef` ist eine Abkürzung für `#if ! defined`.

23.6 `#error` und `#warning`

`#error` gibt eine Fehlermeldung während des Compilerlaufs aus und bricht den Übersetzungsvorgang ab, z.B.:

```
1. error Dieser Quellcode-Abschnitt sollte nicht mit compiliert werden!
```

`#warning` ist ähnlich wie `#error`, mit dem Unterschied, dass das Kompilieren nicht abgebrochen wird. Es ist allerdings nicht Teil von ISO-C++, auch wenn die meisten Compiler es unterstützen. Meistens wird es zum Debuggen eingesetzt:

```
1. warning Dieser Quellcode-Abschnitt muss noch überarbeitet werden.
```

23.7 `#line`

Setzt den Compiler-internen Zeilenzähler auf den angegebenen Wert, z.B.:

```
1. line 100
```

23.8 #pragma

Das #pragma-Kommando ist vorgesehen, um eine Reihe von Compiler-spezifischen Anweisungen zu implementieren, z.B.:

```
1. pragma comment(lib, "advapi32.lib")
```

Kennt ein bestimmter Compiler eine Pragma-Anweisung nicht, so gibt er üblicherweise eine Warnung aus, ignoriert diese nicht für ihn vorgesehene Anweisung aber ansonsten.

Um z.B. bei MS VisualC++ eine "störende" Warnung zu unterdrücken, gibt man folgendes an:

```
1. pragma warning( disable: 4010 ) // 4010 ist Nummer der Warnung
```

23.9 Vordefinierte Präprozessor-Variablen

- __LINE__: Zeilennummer
- __FILE__: Dateiname
- __DATE__: Datum des Präprozessoraufrufs im Format Monat/Tag/Jahr
- __TIME__: Zeit des Präprozessoraufrufs im Format Stunden:Minuten:Sekunden
- __cplusplus: Ist nur definiert, wenn ein C++-Programm verarbeitet wird

Kapitel 24

Headerdateien

Sie haben bereits mit 2 Headerdateien Bekanntschaft gemacht: `iostream` und `string`. Beide sind so genannte Standardheader, das heißt, sie sind in der Standardbibliothek jedes Compilers enthalten.

24.1 Was ist eine Headerdatei?

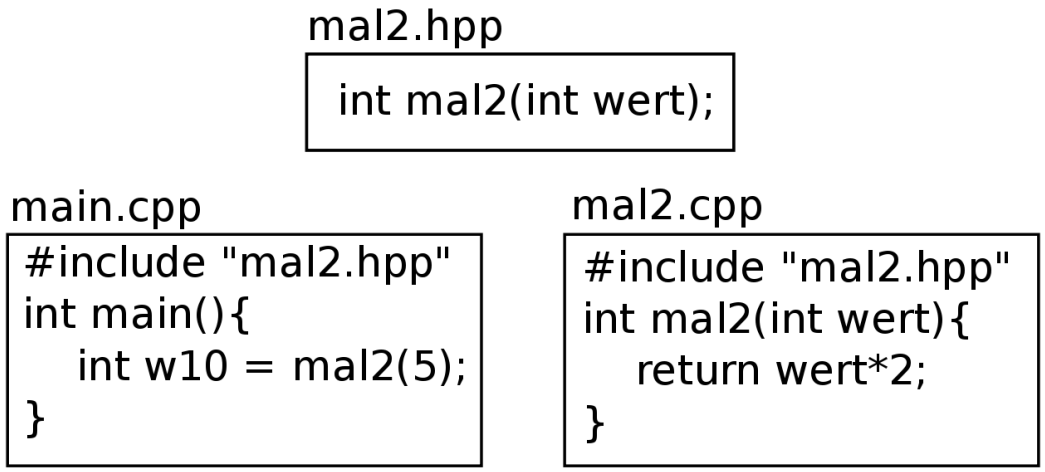

Abbildung 1

Headerdateien sind gewöhnliche C++-Dateien, die im Normalfall Funktionsdeklarationen und ähnliches enthalten. Sicher werden Sie sich erinnern: Deklarationen machen dem Compiler bekannt, wie etwas benutzt wird. Wenn Sie also eine

Headerdatei einbinden und darin eine Funktionsdeklaration steht, dann weiß der Compiler wie die Funktion aufgerufen wird. Der Compiler weiß zu diesem Zeitpunkt nicht, was die Funktion tut, aber das ist auch nicht nötig um sie aufrufen zu können.

Das einbinden einer Headerdatei erfolgt über die Präprozessordirektive `#include`. Der Code in der Datei die mit `include` referenziert wird, wird vom Präprozessor einfach an der Stelle eingefügt, an der das `include` stand.

In der nebenstehenden Darstellung wird die Headerdatei `mal2.hpp` von den Quelldateien `main.cpp` und `mal2.cpp` eingebunden. Um dieses Beispiel zu übersetzen, müssen die Dateien alle im gleichen Verzeichnis liegen. Sie übergeben die Quelldateien an einen Compiler und rufen anschließend den Linker auf, um die entstanden Objektdateien zu einem Programm zu binden.

Hinweis
Falls Sie mit der GCC arbeiten, beachten Sie bitte, dass g++ sowohl Compiler als auch Linker ist. Wenn Sie nur die beiden Quelldateien ohne weitere Parameter angeben, wird nach dem Kompilieren automatisch gelinkt. Der zweite Aufruf entfällt somit. Einige Informationen zu Compilern und Linkern finden Sie übrigens im Kapitel Compiler

24.2 Namenskonventionen

Übliche Dateiendungen für C++-Quelltexte sind „.cpp“ oder „.cc“. Headerdateien haben oft die Endungen „.hpp“, „.hh“ und „.h“. Letztere ist allerdings auch die gebräuchige Dateiendung für C-Header, weshalb zugunsten besserer Differenzierung empfohlen wird, diese nicht zu benutzen.

Die Standardheader von C++ haben überhaupt keine Dateiendung, wie Sie vielleicht schon anhand der beiden Headerdateien `iostream` und `string` erraten haben. In der Anfangszeit von C++ endeten Sie noch auf „.h“, inzwischen ist diese Notation aber nicht mehr gültig, obwohl sie immer noch von vielen Compilern unterstützt wird. Der Unterschied zwischen „iostream“ und „iostream.h“ besteht darin, dass in ersterer Headerdatei alle Deklarationen im Standardnamespace `std` vorgenommen werden.

Prinzipiell kommt es nicht darauf an, welche Dateiendungen Sie ihren Dateien geben, dennoch ist es sinnvoll, eine übliche Endung zu verwenden, wenn auch nur, um einer möglichen Verwechslungsgefahr vorzubeugen. Wenn Sie sich ein-

mal für eine Dateiendung entschieden haben, ist es vorteilhaft, darin eine gewisse Konstanz zu bewahren, nicht nur vielleicht aus Gemütlichkeit und Zeiteinsparnis, sondern auch im Sinne des schnellen Wiederfindens. Einige Beispiele für Headerdateiendungen finden Sie, wenn Sie sich einfach mal einige weitverbreite C++-Bibliotheken ansehen. Boost verwendet „.hpp“, wxWidgets nutzt „.h“ und Qt orientiert sich an der Standardbibliothek, hat also gar keine Dateiendung.

24.3 Schutz vor Mehrfacheinbindung

Da Headerdateien oft andere Headerdateien einbinden, kann es leicht passieren, dass eine Headerdatei mehrfach eingebunden wird. Da viele Header nicht nur Deklarationen, sondern auch Definitionen enthalten, führt dies zu Compiler-Fehlermeldungen, da innerhalb einer Übersetzungseinheit ein Name stets nur genau einmal definiert werden darf (mehrfache Deklarationen, die keine Definitionen sind, sind jedoch erlaubt). Um dies zu vermeiden, wird der Präprozessor verwendet. Am Anfang der Headerdatei wird ein Präprozessor-ifndef ausgeführt, das prüft, ob ein Symbol nicht definiert wurde, ist dies der Fall, wird das Symbol definiert. Am Ende der Headerdatei wird die Abfrage mit einem Präprozessor-endif wieder beendet.

```
1. ifndef _mal2_hpp_

2. define _mal2_hpp_

int mal2(int wert);

1. endif
```

Als Symbol wird üblicherweise aus dem Dateinamen des Headers abgeleitet. In diesem Fall wurde der Punkt durch einen Unterstrich ersetzt und ein weiter Unterstrich vor und nach dem Dateinamen eingefügt. Wenn der Präprozessor nun die Anweisung bekommt, diese Datei in eine andere einzubinden, wird er das anstandslos tun. Findet er eine zweite Anweisung, die Datei einzubinden, wird er alles von `#ifndef` bis `#endif` überspringen, da das Symbol `_mal2_hpp_` ja nun bereits definiert wurde.

Tip
Mehr Informationen über Präprozessor-Anweisungen finden Sie im Kapitel Vorarbeiter des Compilers

24.4 Inline-Funktionen

Im Kapitel „Prozeduren und Funktionen“ haben Sie bereits erfahren was eine Inline-Funktion ist. Das Schlüsselwort `inline` empfiehlt dem Compiler, beim Aufruf einer Funktion den Funktionsrumpf direkt durch den Funktionsaufruf zu ersetzen, wodurch bei kurzen Funktionen die Ausführungsgeschwindigkeit gesteigert werden kann. Es bewirkt aber noch mehr. Normalerweise darf eine Definition immer nur einmal gemacht werden. Da für das Inlinen einer Funktion aber die Definition bekannt sein muss, gibt es für Inline-Funktionen eine Ausnahme: Sie dürfen beliebig oft definiert werden, solange alle Definitionen identisch sind. Deshalb dürfen (und sollten) Inline-Funktionen in den Headerdateien definiert werden, ohne dass sich der Linker später über eine mehrfache Definition in verschiedenen Objektdateien beschweren wird.

Hinweis
Inline-Funktionen dürfen auch in cpp-Dateien definiert werden. Allerdings können sie dann auch nur innerhalb der Objektdatei, die aus der cpp-Datei erzeugt wird, geinlinet werden und das ist in aller Regel nicht beabsichtigt.

Kapitel 25

Das Klassenkonzept

Klassen sind ein wesentlicher Bestandteil der objektorientierten Programmierung. Objektorientiert zu programmieren heißt, Daten und die Funktionen, die darauf angewendet werden, möglichst dicht zusammen zu bringen. Eine Klasse tut genau das: sie beinhaltet Daten und Funktionen. Nach außen hin können die Daten in der Regel nicht direkt zugegriffen werden. Die Verarbeitung der Daten erfolgt über Funktionen der Klasse.

Korrekterweise enthält eigentlich nicht die Klasse die Daten, sondern die Objekte, die von dieser Klasse erzeugt werden. Die Klasse selbst beschreibt lediglich, welche Daten ein Objekt enthalten kann und die Funktionen, die darauf angewendet werden können. Eine Klasse ist also vergleichbar mit einem Datentyp, ein Objekt entspricht dann einer Variablen dieses Datentyps. Tatsächlich werden die Begriffe üblicherweise synonym verwendet.

25.1 Ein eigener Datentyp

Nun wird es aber Zeit, dass wir auch mal eine eigene Klasse schreiben.

```
1. include <iostream>


class Auto{
public:
    Auto(int tankgroesse, float tankinhalt, float verbrauch);
    void info()const;
    bool fahren(int km);
```

```
    void tanken(float liter);
private:
    int   m_tankgroesse;
    float m_tankinhalt;
    float m_verbrauch;
};
Auto::Auto(int tankgroesse, float tankinhalt, float verbrauch):
    m_tankgroesse(tankgroesse),
    m_tankinhalt(tankinhalt),
    m_verbrauch(verbrauch)
    {}
void Auto::info()const{
    std::cout << "In den Tank passen " << m_tankgroesse << " Liter Treibstoff.\n";
    std::cout << "Aktuell sind noch " << m_tankinhalt << " Liter im Tank.\n";
    std::cout << "Der Wagen verbraucht " << m_verbrauch << " Liter pro 100km.\n";
    std::cout << std::endl;
}
bool Auto::fahren(int km){
    std::cout << "Fahre " << km << "km.\n";
    m_tankinhalt -= m_verbrauch*km/100;
    if(m_tankinhalt < 0.0f){
        m_tankinhalt = 0.0f;
        std::cout << "Mit dem aktuellen Tankinhalt schaffen Sie die Fahrt leider nicht.\n";
        std::cout << "Der Wagen ist unterwegs liegengeblieben, Zeit zu tanken!\n";
    }
    std::cout << std::endl;
}
void Auto::tanken(float liter){
    std::cout << "Tanke " << liter << " Liter.\n";
    m_tankinhalt += liter;
    if(m_tankinhalt > m_tankgroesse){
        m_tankinhalt = m_tankgroesse;
        std::cout << "Nicht so übereifrig! Ihr Tank ist jetzt wieder voll.\n";
        std::cout << "Sie haben aber einiges daneben gegossen!\n";
    }
    std::cout << std::endl;
}
```

Diese Klasse demonstriert vieles, was Sie im Laufe dieses Abschnittes noch kennenlernen werden. Für den Moment sollten Sie wissen, dass diese Klasse 3 ver-

schiedene Daten beinhaltet. Diese Daten sind die 3 Variablen deren Namen mit `m_` beginnen. Vier Funktionen arbeiten auf diesen Daten.

Das `m_` steht für Member oder Mitglied, es ist allerdings lediglich eine Möglichkeit, eine Variable als Mitglied einer Klasse zu kennzeichnen. Innerhalb einer Klassenfunktion kann man so schnell erkennen, welche Variablen Membervariablen sind und welche Parameter oder lokale Variablen. Eine Membervariable kann auch jeden anderen gültigen Variablennamen haben. Klassenfunktionen werden auch oft als Memberfunktionen oder schlicht als Methoden bezeichnet. Im Nachfolgendem wird hierfür immer der Begriff „Methode“ verwendet.

Sie haben nun gesehen, wie die Klasse aufgebaut ist, und in den folgenden Kapiteln wird dieser Aufbau genauer erläutert. Jetzt sollen Sie jedoch erst einmal den Vorteil einer Klasse verstehen, denn um eine Klasse zu benutzen, müssen Sie keine Ahnung haben, wie diese Klasse intern funktioniert.

```
int main(){
    Auto wagen(80, 60.0f, 5.7);
    wagen.info();
    wagen.tanken(12.4f);
    wagen.info();
    wagen.fahren(230);
    wagen.info();
    wagen.fahren(12200);
    wagen.info();
    wagen.tanken(99.0f);
    wagen.info();
}
```

Ausgabe

```
In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 60 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100km.
Tanke 12.4 Liter.
In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 72.4 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100km.
Fahre 230km.
In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 59.29 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100km.
Fahre 12200km.
```

```
Mit dem aktuellen Tankinhalt schaffen Sie die Fahrt leider nicht.
Der Wagen ist unterwegs liegengeblieben, Zeit zu tanken!
In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 0 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100km.
Tanke 99 Liter.
Nicht so übereifrig! Ihr Tank ist jetzt wieder voll.
Sie haben aber einiges daneben gegossen!
In den Tank passen 80 Liter Treibstoff.
Aktuell sind noch 80 Liter im Tank.
Der Wagen verbraucht 5.7 Liter pro 100km.
```

In der ersten Zeile von `main()` wird ein `Auto`-Objekt mit dem Namen `wagen` erstellt. Anschließend werden Methoden dieses Objekts aufgerufen, um die Daten darin zu verwalten. Von den Daten innerhalb des Objekts kriegen Sie beim Arbeiten mit dem Objekt überhaupt nichts mit. Lediglich die Ausgabe verrät, dass die drei Methoden untereinander über diese Daten „kommunizieren“.

Die vierte Methode (jene, die mit dem Klassennamen identisch ist) wird übrigens auch aufgerufen. Gleich in der ersten Zeile von `main()` wird diese Methode genutzt, um das Objekt zu erstellen. Daher bezeichnet man diese Methode auch als „Konstruktor“, dazu aber später mehr.

Kapitel 26

Erstellen und Zerstören

In C++-Klassen gibt es zwei besondere Arten von Methoden: Konstruktoren und den Destruktor. Ein Konstruktor wird beim Anlegen eines Objektes ausgeführt, der Destruktor vor der „Zerstörung“ desselben. Der Name des Konstruktors ist immer gleich dem Klassennamen, der Destruktor entspricht ebenfalls dem Klassennamen, jedoch mit einer führendem Tilde (`~`).

Konstruktoren und Destruktoren haben keinen Rückgabetyp, auch nicht `void`. Der Konstruktor kann nicht als Methode aufgerufen werden, beim Destruktor ist dies hingegen möglich (aber nur selten nötig, dazu später mehr).

26.1 Konstruktor

Jede Klasse hat einen oder mehrere Konstruktoren. Ein solcher Konstruktor dient zur Initialisierung eines Objektes. Im Folgenden wird eine Klasse `Bruch` angelegt, die über den Konstruktor die Membervariablen `m_zaehler` und `m_nenner` initialisiert.

```
class Bruch{
public:
    Bruch(int z, int n):
        m_zaehler(z), // Initialisierung von m_zaehler
        m_nenner(n)   // Initialisierung von m_nenner
        {}
private:
    int m_zaehler;
```

```
    int m_nenner;
};
int main(){
    Bruch objekt(7, 10); // Der Bruch 7/10 oder auch 0.7
}
```

Wie Sie sehen, ist der Methodenrumpf von `Bruch::Bruch` leer. Die Initialisierung findet in den beiden Zeilen über dem Rumpf statt. Nach dem Prototyp wird ein Doppelpunkt geschrieben, darauf folgt eine Liste der Werte, die initialisiert werden sollen. Vielleicht erinnern Sie sich noch, dass es zur Initialisierung zwei syntaktische Varianten gibt:

```
int wert = 8;
int wert(8);
```

Innerhalb der Initialisierungsliste ist nur die zweite Variante zulässig. Auch in der Hauptfunktion `main()` findet eine Initialisierung statt. Wie Sie sehen, ist hier ebenfalls nur die zweite Variante möglich, da `objekt` zwei Werte erwartet.

Hinweis
Beachten Sie, dass die Initialisierung der Variablen einer Klasse in der Reihenfolge erfolgt, in der sie in der Klasse deklariert wurden. Die Initialisierungsreihenfolge ist *unabhängig* von der Reihenfolge, in der sie in der Initialisierungsliste angegeben wurden. Viele Compiler warnen, wenn man so etwas macht, da derartiger Code möglicherweise nicht das tut, was man erwartet.

Natürlich ist es wie bei Funktionen möglich (und in der Regel zu empfehlen), die Methodendeklaration von der Methodendefinition zu trennen. Die Definition sieht genau so aus, wie bei einer normalen Funktion. Der einzige auffällige unterschied besteht darin, dass dem Methodennamen der Klassenname vorangestellt wird, getrennt durch den Bereichsoperator (`::`).

```
class Bruch{
public:
    Bruch(int z, int n);     // Deklaration
private:
    int m_zaehler;
    int m_nenner;
};
```

```
Bruch::Bruch(int z, int n):  // Definition
    m_zaehler(z),            // Initialisierung von m_zaehler
    m_nenner(n)              // Initialisierung von m_nenner
    {}
int main(){
    Bruch objekt(7, 10);     // Der Bruch 7/10 oder auch 0.7
}
```

Wie bereits erwähnt, hat der Konstruktor keinen Rückgabetyp, daher wird auch in der Definition keiner angegeben. Bei den Basisdatentypen ist es bezüglich der Performance übrigens egal, ob Sie diese initialisieren oder zuweisen. Sie könnten also den gleichen Effekt erzielen, wenn Sie statt der Initialisierungsliste eine Zuweisung im Funktionsrumpf benutzen. Allerdings gilt dies wirklich nur für die Basisdatentypen, bei komplexen Datentypen ist die Initialisierung oft deutlich schneller. Außerdem können konstante Variablen ausschließlich über die Initialisierungsliste einen Wert erhalten. Nun aber noch mal ein einfaches Beispiel, in dem der Funktionsrumpf des Konstruktors zur Anfangswertzuweisung dient:

```
class Bruch{
public:
    Bruch(int z, int n = 1); // Deklaration
private:
    int m_zaehler;
    int m_nenner;
};
Bruch::Bruch(int z, int n){  // Definition
    m_zaehler = z;
    m_nenner  = n;
}
```

Wie Sie sehen entfällt der Doppelpunkt, wenn Sie die Initialisierungsliste nicht nutzen. Natürlich können Sie auch Initialisierungsliste und Funktionsrumpf parallel benutzen.

Irgendwann werden Sie sicher einmal in die Verlegenheit kommen ein Array als Membervariable innerhalb Ihrer Klasse zu Deklarieren. Leider gibt es keine Möglichkeit Arrays zu initialisieren, sie müssen immer im Konstruktorrumpf mittels Zuweisung ihren Anfangswert erhalten. Entsprechend ist es auch nicht möglich ein Memberarray mit konstanten Daten zu erstellen. Wenn Sie also zwingend eine Initialisierung benötigen, müssen Sie wohl oder übel Einzelvariablen erstellen.

26.1.1 Defaultparameter

Sie können einem Konstruktor ebenso Defaulparameter vorgeben wie einer gewöhnlichen Funktion. Die Syntax ist hierbei identisch.

```
class Bruch{
public:
    Bruch(int z, int n = 1); // Deklaration
private:
    int m_zaehler;
    int m_nenner;
};
Bruch::Bruch(int z, int n):  // Definition
    m_zaehler(z),
    m_nenner(n)
    {}
```

26.1.2 Mehrere Konstruktoren

Auch das Überladen des Konstruktors funktioniert wie das Überladen einer Funktion. Deklarieren Sie mehrere Prototypen innerhalb der Klasse und schreiben für jeden eine Definition:

```
class Bruch{
public:
    Bruch(int z);        // Deklaration
    Bruch(int z, int n); // Deklaration
private:
    int m_zaehler;
    int m_nenner;
};
Bruch::Bruch(int z):        // Definition
    m_zaehler(z),
    m_nenner(1)
    {}
Bruch::Bruch(int z, int n): // Definition
    m_zaehler(z),
```

```
    m_nenner(n)
    {}
```

Wenn mehrere Konstruktoren das gleiche tun, ist es oft sinnvoll, diese gleichen Teile in eine eigene Methode (üblicherweise mit dem Namen `init()`) zu schreiben. Das ist kürzer, übersichtlicher, meist schneller und hilft auch noch bei der Vermeidung von Fehlern. Den wenn Sie den Code nachträglich ändern, dann müssten Sie diese Änderungen sonst für jeden Konstruktor vornehmen. Nutzen Sie eine `init()`-Methode, müssen Sie denn Code nur innerhalb von dieser einmal ändern.

```
class A{
public:
    A(double x);
    A(int x);
private:
    void init();
    int m_x;
    int m_y;
};
A::A(double x):
    m_x(x) {
    init();
    }
A::A(int x):
    m_x(x) {
    init();
    }
void A::init(){
    m_y=7000;
}
```

Dieses Beispiel ist zugegebenermaßen ziemlich sinnfrei, aber das Prinzip der `init()`-Funktion wird deutlich. Wenn Sie sich nun irgendwann entscheiden, `m_y` als Anfangswert `3333` zuzuweisen, dann müssen Sie dies nur in der `init()`-Funktion ändern und nicht in jedem einzelnen Konstruktor.

26.1.3 Standardkonstruktor

Als Standardkonstruktor bezeichnet man einen Konstruktor, der keine Parameter erwartet. Ein Standardkonstruktor für unsere `Bruch`-Klasse könnte zum Beispiel folgendermaßen aussehen:

```
class Bruch{
public:
    Bruch();      // Deklaration Standardkonstruktor
private:
    int m_zaehler;
    int m_nenner;
};
Bruch::Bruch():   // Definition Standardkonstruktor
    m_zaehler(0),
    m_nenner(1)
    {}
```

Natürlich könnten wir in diesem Beispiel den Konstruktor überladen, um neben dem Standardkonstruktor auch die Möglichkeiten zur Initialisierung zu haben. In diesem Beispiel bietet es sich jedoch an, dafür Defaultparameter zu benutzen. Das folgende kleine Beispiel erlaubt die Initialisierung mit einer ganzen Zahl und mit einer gebrochenen Zahl (also zwei Ganzzahlen: Zähler und Nenner). Außerdem wird auch gleich noch der Standardkonstruktor bereitgestellt, welcher den Bruch mit 0 initialisiert, also den Zähler auf Null und den Nenner auf Eins setzt.

```
class Bruch{
public:
    // Erlaubt 3 verschiedene Aufrufmöglichkeiten, darunter die des Standardkonstruktors
    Bruch(int z = 0, int n = 1);
private:
    int m_zaehler;
    int m_nenner;
};
Bruch::Bruch(int z, int n): // Definition des Konstruktors
    m_zaehler(z),
    m_nenner(n)
    {}
```

Im Kapitel "Leere Klassen" werden Sie noch einiges mehr über den Standardkonstruktor erfahren.

26.1.4 Kopierkonstruktor

Neben dem Standardkonstruktor hat der Kopierkonstruktor noch eine besondere Bedeutung. Er erstellt, wie der Name bereits andeutet, ein Objekt einer Klasse, anhand eines bereits vorhanden Objekt. Der Parameter des Kopierkonstruktors ist also immer eine Referenz auf ein konstantes Objekt der selben Klasse. Der Kopierkonstruktor unserer `Bruch`-Klasse hat folgende Deklaration.

```
Bruch::Bruch(Bruch const& bruch_objekt);
```

Wenn wir keinen eigenen Kopierkonstruktor schreiben, erstellt der Compiler einen für uns. Dieser implizite Kopierkonstruktor initialisiert alle Membervariablen mit den entsprechenden Werten der Membervariablen im übergebenen Objekt. Für den Moment ist diese vom Compiler erzeugte Variante ausreichend. Später werden Sie Gründe kennenlernen, die es notwendig machen, einen eigenen Kopierkonstruktor zu schreiben. Wenn Sie die Nutzung des Kopierkonstruktors verbieten wollen, dann schreiben Sie seine Deklaration in den `private`-Bereich der Klassendeklaration. Das bewirkt einen Kompilierfehler, wenn jemand versucht den Kopierkonstruktor aufzurufen.

Neben dem Kopierkonstruktor erzeugt der Compiler übrigens auch noch eine Kopierzuweisung; auch zu dieser werden Sie später noch mehr erfahren. Auch diese können Sie verbieten, indem Sie sie im `private` deklarieren. Wie dies geht, erfahren Sie im Kapitel zu Operatorüberladung.

26.2 Destruktor

Im Gegensatz zum Konstruktor, gibt es beim Destruktor immer nur einen pro Klasse. Das liegt daran, dass ein Destruktor keine Parameter übergeben bekommt. Sein Aufruf erfolgt in der Regel implizit durch den Compiler bei der Zerstörung eines Objektes. Für den Anfang werden Sie mit dem Destruktor wahrscheinlich nicht viel anfangen können, da es mit den Mitteln, die Sie bis jetzt kennen, kaum nötig werden kann, dass bei der Zerstörung eines Objektes Aufräumungsarbei-

ten ausgeführt werden. Das folgende kleine Beispiel enthält einen Destruktor, der einfach gar nichts tut:

```
class A{
public:
    ~A(); // Deklaration Destruktor
};
A::~A(){  // Definition Destruktor
    // Hier könnten "Aufräumungsarbeiten" ausgeführt werden
}
```

Wenn Sie den Abschnitt über Speicherverwaltung gelesen haben, werden Sie wissen, wie nützlich der Destruktor ist. Im Moment reicht es, wenn Sie mal von ihm gehört haben. Auch über den Destruktor werden Sie im Kapitel "Leere Klassen" weitere Informationen erhalten.

26.3 Beispiel mit Ausgabe

Um noch einmal deutlich zu machen, an welchen Stellen Konstruktor und Destruktor aufgerufen werden, geben wir einfach innerhalb der Methoden eine Nachricht aus:

```
    1. include <iostream>


class A{
public:
    A();          // Deklaration Konstruktor
    ~A();         // Deklaration Destruktor
    void print(); // Deklaration einer Methode
};
A::A(){           // Definition Konstruktor
    std::cout << "Innerhalb des Konstruktors" << std::endl;
}
A::~A(){          // Definition Destruktor
    std::cout << "Innerhalb des Destruktors" << std::endl;
}
void A::print(){  // Definition der print()-Methode
    std::cout << "Innerhalb der Methode print()" << std::endl;
```

```
}
int main(){
    A objekt;       // Anlegen des Objekts == Konstruktoraufruf
    objekt.print(); // Aufruf der Methode print()
}                   // Ende von main(), objekt wird Zerstört == Destruktoraufruf
```

Ausgabe

```
Innerhalb des Konstruktors
Innerhalb der Methode print()
Innerhalb des Destruktors
```

Kapitel 27

Privat und öffentlich

Mit den Schlüsselwörtern `public` und `private` wird festgelegt, von wo aus auf ein Mitglied einer Klasse zugegriffen werden kann. Auf Member, die als `public` deklariert werden, kann von überall aus zugegriffen werden, sie sind also *öffentlich* verfügbar. `private`-deklarierte Member lassen sich nur innerhalb der Klasse ansprechen, also nur innerhalb von Methoden derselben Klasse. Typischerweise werden Variablen `private` deklariert, während Methoden `public` sind. Eine Ausnahme bilden Hilfsmethoden, die gewöhnlich im `private`-Bereich deklariert sind, wie etwa die `init()`-Methode, die von verschieden Konstruktoren aufgerufen wird, um Codeverdopplung zu vermeiden.

Standardmäßig sind die Member einer Klasse `private`. Geändert wird der Sichtbarkeitsbereich durch die beiden Schlüsselwörter gefolgt von einem Doppelpunkt. Alle Member die darauffolgend Deklarierten werden, fallen in den neuen Sichtbarkeitsbereich.

```
class A{
    // private Member
public:
    // öffentliche Member
private:
    // private Member
};
```

Zwischen den Sichbarkeitsbereichen kann somit beliebig oft gewechselt werden. Der implizit private Bereich sollte aus Gründen der Übersichtlichkeit nicht genutzt werden. In diesem Buch folgen wir der Regel, als erstes die öffentlichen Member

zu deklarieren und erst danach die privaten. Die umgekehrte Variante findet jedoch ebenfalls eine gewisse Verbreitung. Wie üblich gilt, entscheiden Sie sich für eine Variante und wenden Sie diese konsequent an.

Kapitel 28

Klassen und const

Auch für Klassen gilt üblicherweise: Verwenden Sie `const` wann immer es möglich ist. Wie Sie bereits wissen, sollte `const` immer verwendet werden, wenn eine Variable nach der Initialisierung nicht mehr verändert werden soll.

Da Klassen Datentypen sind, von denen Instanzen (also Variablen) erstellt werden können, ist es natürlich möglich ein Objekt zu erstellen, das konstant ist. Da der Compiler jedoch davon ausgehen muss, dass jede Methode der Klasse die Daten (und somit das Objekt) verändert, sind Methodenaufrufe für konstante Objekte nicht möglich. Eine Ausnahme bilden jene Methoden, die ebenfalls als `const` gekennzeichnet sind. Ein solche Methode kann zwar problemlos Daten aus dem Objekt lesen, aber niemals darauf schreiben und auch für Dritte keine Möglichkeit bereitstellen, objektinterne Daten zu verändern.

28.1 Konstante Methoden

Unsere Beispielklasse `Auto` enthält die konstante Methode `info()`, sie greift nur lesend auf die Membervariablen zu, um ihre Werte auszugeben. Wenn Sie ein konstantes `Auto` Objekt erstellen, können Sie `info()` problemlos aufrufen. Versuchen Sie jedoch `fahren()` oder `tanken()` aufzurufen, wird Ihr Compiler dies mit einer Fehlermeldung quittieren.

```
class Auto{
// ...
    void info()const;
    bool fahren(int km);
```

```
    void tanken(float liter);
// ...
};
```

Wie Sie an diesem Beispiel sehen, lässt sich eine Methode als konstant auszeichnen, indem nach der Parameterliste das Schlüsselwort `const` angegeben wird. Diese Auszeichnung folgt also auch der einfachen Regel: `const` steht immer rechts von dem, was konstant sein soll, in diesem Fall die Methode. Da `const` zum Methodenprototyp zählt, muss es natürlich auch bei der Definition der Methode angegeben werden.

Es sei noch einmal explizit darauf hingewiesen, dass sich die Konstantheit einer Methode lediglich auf die Membervariablen der zugehörigen Klasse auswirkt. Es ist problemlos möglich, eine als Parameter übergebene Variable zu modifizieren.

Eine konstante Methode kann ausschließlich andere konstante Methoden des eigenen Objektes aufrufen, denn der Aufruf einer nicht-konstanten Methode könnte ja Daten innerhalb des Objektes ändern.

28.2 Sinn und Zweck konstanter Objekte

Vielleicht haben Sie sich bereits gefragt, wofür es gut sein soll, ein Objekt als konstant auszuzeichnen, wenn der Zweck eines Objektes doch darin besteht, mit den enthaltenen Daten zu arbeiten. Beim Erstellen eines konstanten Objektes können Sie es einmalig über den Konstruktor mit Werten belegen. In diesem Fall haben Sie von einem konstanten Objekt das gleiche, wie bei konstanten Variablen von Basisdatentypen.

Oft ist es jedoch nicht möglich, alle Einstellungen zu einem Objekt über den Konstruktoraufruf festzulegen. Es fördert die Übersichtlichkeit schließlich nicht, wenn man, etwa 20 verschiedene Konstruktoren mit je etwa 50 Parametern hat. Der Ansatz, Klassen so zu gestalten, dass man immer alle Werte über den Konstruktor festlegen kann, hat also seine Grenzen. In diesem Fall hat es einfach keinen Sinn, ein Objekt bei der Erstellung konstant zu machen, denn die Einstellungen werden erst nach dem Erstellen der Objektes vorgenommen.

Wenn Sie ein so erstelltes Objekt nun allerdings an eine Funktion übergeben und diese Funktion keine Veränderungen an dem Objekt vornimmt, ist die Wahrscheinlichkeit groß, dass der Parameter ein konstantes Objekt ist. Innerhalb einer solchen Funktion wäre das Objekt also konstant.

28.3 Zugriffsmethoden

Zugriffsmethoden sollten eigentlich vermieden werden, aber manchmal sind sie nützlich. Eine Zugriffsmethode mach nichts anderes, als eine Membervariable lesend oder schreibend zugreifbar zu machen:

```
class A{
public:
    void SetWert(int wert) { m_wert = wert; }
    int  GetWert()const    { return m_wert; }
private:
    int m_wert;
};
```

Get-Methoden sind immer konstant, da sie den Wert ja nur lesend zugreifbar machen sollen. Eine Set-Methode kann dagegen nie mit einem konstanten Objekt benutzt werden. Im Normalfall sollten Sie jedoch keine „Getter“ oder „Setter“ benötigen, wenn doch, müssen Sie sich Gedanken darüber machen, ob das Objekt die Logik möglicherweise nicht ausreichend kapselt.

Solche Einzeiler werden normalerweise einfach direkt in die Funktionsdeklaration geschrieben, dadurch sind Sie auch gleich automatisch als `inline` ausgezeichnet. Dazu müssen Sie nur das Semikolon durch den Funktionsrumpf ersetzen. Sollten Sie dennoch lieber eine eigene Definition für solche Methoden machen wollen, dann achten Sie darauf, diese als Definition als `inline` zu kennzeichnen. Falls Sie mit Headerdateien arbeiten, dann beachten Sie, dass der Funktionsrumpf bei `inline`-Methoden während des Kompilierens bekannt sein muss. Die Definition muss also mit in die Headerdatei, nicht wie gewöhnlich in die Quelldatei.

Kapitel 29

Überladen...

Sie haben bereits gelernt, dass Funktionen überladen werden können, indem für den gleichen Funktionsnamen mehrere Deklarationen mit verschiedenen Parametern gemacht werden. Auch bei Klassenkonstruktoren haben Sie Überladung bereits kennengelernt. Für gewöhnliche Memberfunktionen ist eine Überladung ebenfalls nach den Ihnen bereits bekannten Kriterien möglich möglich. Zusätzlich können Sie Memberfunktionen aber anhand des eben vorgestellten `const`-Modifizierers überladen.

```
class A{
public:
    void methode();      // Eine Methode
    void methode()const; // Die überladene Version der Methode für konstante Objekte
};
```

Natürlich können auch Methoden mit Parametern auf diese Weise überladen werden. Die nicht-konstante Version wird immer dann aufgerufen, wenn Sie mit einem nicht-konstanten Objekt der Klasse arbeiten. Analog dazu wird die konstante Version aufgerufen, wenn Sie mit einem konstanten Objekt arbeiten. Wenn Sie nur eine konstante Version deklarieren, wird immer diese Aufgerufen.

Sinnvoll ist diese Art der Überladung vor allem dann, wenn Sie einen Zeiger oder eine Referenz auf etwas innerhalb des Objekts zurückgeben. Wie Sie sich sicher erinnern, kann eine Überladung nicht anhand des Rückgabetyps eine Funktion (oder Methode) gemacht werden. Das folgende Beispiel wird Ihnen zeigen, wie Sie eine `const`-Überladung nutzen können, um direkte Manipulation von Daten innerhalb eines Objekts nur für nicht-konstante Objekte zulassen.

```
    1. include <iostream>


class A{
public:
    A():m_b(7) {} // m_b mit 7 initialisieren
    int&       B()      { return m_b; } // Zugriff lesend und schreibend
    int const& B()const { return m_b; } // Zugriff nur lesend
private:
    int m_b; // Daten
};
int main(){
    A       objekt;       // Ein Objekt von A
    A const objekt_const; // Ein konstantes Objekt von A
    std::cout << objekt.B()       << std::endl; // Gibt 7 aus
    std::cout << objekt_const.B() << std::endl; // Gibt 7 aus
    objekt.B() = 9; // setzt den Wert von m_b auf 9
    // objekt_const.B() = 9; // Produziert einen Kompilierfehler
    std::cout << objekt.B()       << std::endl; // Gibt 9 aus
    std::cout << objekt_const.B() << std::endl; // Gibt 7 aus
}
```

Im Kapitel über Operatorüberladung werden Sie noch ein Beispiel zu dieser Technik kennenlernen, welches in der Praxis oft zu sehen ist.

29.1 Code-Verdopplung vermeiden

Im Beispiel von eben geben die beiden Funktionen lediglich eine Referenz auf eine Membervariable innerhalb des Objekts zurück. In der Regel wird eine solche Funktionen natürlich noch etwas mehr tun. Daher währe es nötig, zwei Funktionen zu schreiben, die den gleichen Code enthalten. Das wiederum ist ausgesprochen schlechter Stil. Stellen Sie sich nur vor, Sie möchten die Funktion später aus irgendwelchen Gründen ändern, dann müssten Sie alle Änderungen an zwei Stellen im Code vornehmen.

Daher ist es sinnvoll, wenn eine Variante die andere aufruft. Hiefür sind einige Tricks nötig, da Sie einer der beiden Varianten beibringen müssen, eine Methode aufzurufen, die eigentlich nicht zum `const`-Modifizierer des aktuellen Objekts passt. Die konstante Variante verspricht niemals eine Änderung am Objekt vor-

zunehmen, sie ist in ihren Möglichkeiten also stärker eingeschränkt. Die nicht konstante Version darf hingegen alles, was auch die konstante Version darf. Somit ist es sinnvoll, die konstante Version von der nicht-konstanten aufrufen zu lassen.

Um nun der nicht-konstanten Version beizubringen, dass sie ihr konstantes Äquivalent aufrufen soll, müssen wir zunächst einmal aus dem aktuellen Objekt ein konstantes Objekt machen. Jedes Klasse enthält eine spezielle Variable, den sogenannten `this`-Zeiger, der innerhalb einer Membervariable einen Zeiger auf das aktuelle Objekt repräsentiert. Diesen `this`-Zeiger casten wir in einen Zeiger auf ein konstantes Objekt.

```
// nicht-konstante Version von B()
int A::B(){
    A const* objekt_const = static_cast< A const* >(this); // Konstantheit dazu casten
```

Nun haben wir einen Zeiger auf das Objekt, über den wir nur konstante Methoden aufrufen können. Das Problem ist nun, dass die aufgerufene Methode natürlich auch eine Referenz auf eine konstante Variable aufruft.

```
   int const& result = objekt_const->B(); // konstante Methodenversion aufrufen
```

Da wir ja wissen, dass das aktuelle Objekt eigentlich gar nicht konstant ist, können wir die Konstantheit für die zurückgegebene Referenz guten Gewissens entfernen. Allerdings ist der `static_cast`, den Sie bereits kennen, nicht dazu in der Lage. Um Konstantheit zu entfernen, benötigen Sie den `const_cast`. Beachten Sie jedoch, dass dieser Cast wirklich nur auf Variablen angewendet werden darf, die eigentlich nicht konstant sind!

```
   return const_cast< int& >(result); // Konstantheit vom Rückgabewert wegcasten
}
```

Wenn Sie diese Anweisungen in einer Zusammenfassen, sieht Ihre Klasse nun folgendermaßen aus.

```
class A{
public:
    // m_b mit 7 initialisieren
    A():m_b(7) {}
```

```
    // Ruft B()const auf
    int&       B()      { return const_cast< int& >( static_cast< A const* >(this)->B() ); }
    int const& B()const { return m_b; }
private:
    int m_b; // Daten
};
```

Wie schon gesagt, sieht diese Technik in unserem Beispiel überdimensioniert aus. Aber auch an dieser Stelle möchte ich Sie auf das Beispiel im Kapitel zur Operatorüberladung verweisen, wo sie, aufgrund der umfangreicheren konstanten Version, bereits deutlich angenehmer erscheint. Mit Performanceeinbußen haben Sie an dieser Stelle übrigens nicht zu rechnen. Ihr Compiler wird die casting-Operationen wegoptimieren.

Kapitel 30

Call by reference

Im Gegensatz zu den vorangegangen Kapiteln dieses Abschnitts, geht es diesmal nicht darum, wie man Objekte aufbaut, sondern wie man mit ihnen arbeitet. Im Kapitel über Funktionen hatten Sie schon ersten Kontakt mit der Wertübergabe als Referenz. Wie dort bereits erwähnt, ist es für Klassenobjekte effizienter, sie als Referenz an eine Funktion zu übergeben.

Bei der Übergabe einer Variablen als Wert muss von dieser Variable erst eine Kopie angefertigt werden. Zusätzlich führt der Kopierkonstruktor an dieser Stelle möglicherweise noch irgendwelche zusätzlichen Operationen aus, die Zeit kosten. Bei einer Übergabe als Referenz muss hingegen nur die Speicheradresse des Objekts kopiert werden. Wenn wir dann noch dafür sorgen, dass die Referenz auf ein konstantes Objekt verweist, haben wir eine fast kostenlose Übergabe und gleichzeitig die Sicherheit, dass die übergebene Variable innerhalb der Funktion nicht verändert wird.

```
class A{
public:
    int a, b, c, d;
};
class B{
public:
    // Das übergebene A-Objekt ist innerhalb der Methode konstant
    void methode(A const& parameter_name);
};
```

Die Methode von `B` kann auf die 4 Variablen von `A` lesend zugreifen, sie aber nicht verändern. Die Übergabe als Wert ist dann sinnvoll, wenn innerhalb einer Methode ohnehin eine Kopie der Variablen benötigt wird.

```
class B{
public:
    // Es wird eine Kopie des A-Objekts übergeben
    void methode(A parameter_name);
};
```

Nun können auf das A-Objekt beliebige lesende oder schreibende Operationen angewendet werden. Da sie auf einer Kopie ausgeführt werden, bleibt auch hier das Originalobjekt unverändert, aber eben zu dem Preis, dass zusätzlich Zeit und Speicherplatz benötigt werden, um eine Kopie des Objekts zu erstellen. In den meisten Fällen ist es nicht nötig, innerhalb einer Methode, Schreiboperationen auszuführen. Verwenden Sie daher nach Möglichkeit die „Call by reference“-Variante.

Bei Rückgabewerten sollten Sie natürlich auf Referenzen verzichten, es sei den, Sie wissen wirklich was Sie tun. Andernfalls kann es schnell passieren, dass Sie eine Referenz auf eine Variable zurückgeben, die außerhalb der Methode gar nicht mehr Existiert. Das wiederum führt zufallsbedingt zu Laufzeitfehlern und Programmabstürzen.

Kapitel 31

Operatoren überladen

Im Kapitel über Zeichenketten haben Sie gelernt, das es sich bei `std::string` um eine Klasse handelt. Dennoch wahr es Ihnen möglich, mehrere `std::string`-Objekte über den +-Operator zu verknüpfen oder einen String mittels = bzw. += an einen `std::string` zuzuweisen bzw. anzuhängen. Der Grund hierfür ist, das diese Operatoren für die `std::string`-Klasse überladen wurden. Das heißt ihnen wurde in Zusammenhang mit der Klasse `std::string` eine neue Bedeutung zugewiesen.

Um Operatoren überladen zu können müssen Sie zunächst einmal wissen, das diese in C++ im Prinzip nur eine spezielle Schreibweise für Funktionen sind. Das folgende kleine Beispiel demonstriert den Aufruf der überladenen `std::string` in ihrer Funktionsform:

```
1. include <iostream>
2. include <string>


int main(){
    std::string a = "7", b = "3", c;
    c = a + b;
    std::cout << c << std::endl;
    c.operator=(operator+(a,b));
    std::cout << c << std::endl;
}
```

Wie Ihnen beim Lesen des Beispiels vielleicht schon aufgefallen ist, wurde der Zuweisungsoperator als Methode von `std::string` überladen. Der Verkettungs-

operator ist hingegen als gewöhnliche Funktion überladen. Dieses Vorgehen ist üblich, aber nicht zwingend. Die Deklarationen der beiden Operatoren sehen folgendermaßen aus:

```
class std::string{
    // Zuweisungsoperator als Member von std::string
    std::string& operator=(std::string const& assign_string);
};
// Verkettungsoperator als globale Funktion
std::string operator+(std::string const& lhs, std::string const& rhs);
```

Da es sich bei `std::string` um ein `typedef` von eine Templateklasse handelt, sehen die Deklarationen in der C++-Standardbibliothek noch etwas anders aus, das ist für unser Beispiel aber nicht relevant. Auf die Rückgabetypen der verschiedenen Operatoren wird später in diesem Kapitel noch näher eingegangen.

Wenn Sie den Zuweisungsoperator als globale Funktion überladen wollten, müssten Sie ihm zusätzlich als ersten Parameter eine Referenz auf ein `std::string`-Objekt übergeben. Würden Sie den Verkettungsoperator als Methode der Klasse deklarieren wollen, entfiele der erste Parameter. Er würde durch das Objekt ersetzt, für das der Operator aufgerufen wird. Da die Operatorfunktion nichts an dem Objekt ändert, für das sie aufgerufen wird, ist sie außerdem konstant. Allerdings hätten Sie an dieser Stelle ein Problem, sobald Sie einen `std::string` zum Beispiel mit einem C-String verknüpfen wollen.

Bei einer Operatorüberladung als Methode ist der erste Operand immer ein Objekt der Klasse. Für den Zuweisungsoperator ist das in Ordnung, da ja immer etwas an einen `std::string` zugewiesen werden soll. Wenn Sie also folgende Operatorüberladungen in der Klasse vornahmen, werden Sie bei der Nutzung der Klasse schnell auf Probleme stoßen.

```
class std::string{
    // Zuweisung eines anderen std::string's
    std::string& operator=(std::string const& assign_string);
    // Zuweisung eines C-Strings
    std::string& operator=(char const assign_string[]);
    // Verkettung mit anderen std::string's
    std::string operator+(std::string const& rhs)const;
    // Verkettung mit C-Strings
    std::string operator+(char const rhs[])const;
```

```
};
int main(){
    std::string std_str_1 = "std_1";
    std::string std_str_2 = "std_2";
    char        c_str_1[80] = "c_str";
    std_str_1 = std_str_2; // Geht
    std_str_1 = c_str_1;   // Geht
//  c_str_1   = std_str_1; // Geht nicht (soll es auch nicht!)
    std_str_1 + std_str_2; // Geht
    std_str_1 + c_str_1;   // Geht
//  c_str_1   + std_str_2; // Geht nicht (sollte es aber)
}
```

Die in der Klasse deklarierten Formen lassen sich problemlos nutzen. Wenn wir aber wollen, dass man auch „C-String + `std::string`“ schreiben kann, müssen wir noch eine weitere Operatorüberladung global deklarieren. Da dies jedoch uneinheitlich aussieht und obendrein die Klassendeklaration unnötig aufbläht, deklariert man alle Überladungen des Verkettungsoperators global. Das Ganze sieht dann folgendermaßen aus.

```
std::string operator+(std::string const& lhs, std::string const& rhs);
std::string operator+(std::string const& lhs, char const rhs[]);
std::string operator+(char const lhs[],       std::string const& rhs);
```

31.1 Definition der überladenen Operatoren

31.2 Codeverdopplung vermeiden

31.3 Ein-/Ausgabeoperatoren überladen

31.4 Spaß mit „falschen“ Überladungen

Mit Hilfe der Operatorüberladung lässt sich einiger Unsinn anstellen. Die folgende kleine Klasse verhält sich wie ein normaler `int`, allerdings rechnet der Plus-

operator minus, und umgekehrt. Gleiches gilt für die Punktoperationen. Natürlich sollten Sie solch unintuitives Verhalten in ernst gemeinten Klassen vermeiden, aber an dieser Stelle demonstriert es noch einmal schön die Syntax zur Überladung.

```
// Zum Überladen von Ein-/Ausgabe
    1. include <ios>


class Int{
public:
    // Konstruktor (durch Standardparameter gleichzeitig Standardkonstruktor)
    Int(int value = 0): value(m_value) {}
    // Überladene kombinierte Rechen-/Zuweisungsoperatoren
    Int const& operator+=(Int const& rhs) { m_value -= rhs.m_value; return *this; }
    Int const& operator-=(Int const& rhs) { m_value += rhs.m_value; return *this; }
    Int const& operator*=(Int const& rhs) { m_value /= rhs.m_value; return *this; }
    Int const& operator/=(Int const& rhs) { m_value *= rhs.m_value; return *this; }
private:
    int m_value;
// Friend-Deklarationen für Ein-/Ausgabe
friend std::ostream& operator<<(std::ostream& os, Int const& rhs);
friend std::istream& operator>>(std::istream& is, Int& rhs);
};
// Überladene Rechenoperatoren rufen kombinierte Versionen auf
Int operator+(Int const& lhs, Int const& rhs) { return Int(lhs) += rhs; }
Int operator-(Int const& lhs, Int const& rhs) { return Int(lhs) -= rhs; }
Int operator*(Int const& lhs, Int const& rhs) { return Int(lhs) *= rhs; }
Int operator/(Int const& lhs, Int const& rhs) { return Int(lhs) /= rhs; }
// Definition Ein-/Ausgabeoperator
std::ostream& operator<<(std::ostream& os, Int const& rhs) { return os << rhs.m_value; }
std::istream& operator>>(std::istream& is, Int& rhs) { return is >> rhs.m_value; }
```

31.5 Präfix und Postfix

Für einige Klassen kann es sinnvoll sein, den Increment- und den Decrement-Operator zu überladen. Da es sich um unäre Operatoren handelt, sie also nur einen Parameter haben, werden sie als Klassenmember überladen. Entsprechend über-

nehmen sie überhaupt keine Parameter mehr, da dieser ja durch das Klassenobjekt, für das sie aufgerufen werden ersetzt wird. Als Beispiel nutzen wir noch einmal unsere Int-Klasse, diesmal aber ohne die Operatoren mit der falschen Funktionalität zu überladen.

```
class Int{
public:
// ...
    Int& operator++() { ++m_value; return *this; }
    Int& operator-() { -m_value; return *this; }
// ...
};
// ...
int main(){
    Int value(5);
    std::cout << value << std::endl;   // value ist 5
    std::cout << ++value << std::endl; // value ist 6
    std::cout << -value << std::endl; // value ist 5
}
```

So weit, so gut, aber wir können nicht `value++` oder `value-` schreiben. Diese Variante überlädt also nur die Präfix-Version der Operatoren, aber wie schreibt man nun die Postfix-Überladung? Wie Sie wissen, kann eine Funktion nur anhand ihrer Parameter oder, im Fall einer Methode, ihres `const`-Qualifizieres überladen werden. Aber weder die Prafix- noch die Postfixvariante übernehmen einen Parameter und auch der `const`-Qualifizier ist in keinem Fall gegeben, weil die Operatoren ja immer das Objekt verändern.

Aus diesem Grund hat man sich eine Kompromisslöschung ausdenken müssen. Die Postfix-Operatoren übernehmen einen `int`, der jedoch innerhalb der Funktionsdefinition nicht verwendet wird. Er dient einzig, um dem Compiler mitzuteilen das es sich um eine Postfix-Operatorüberladung handelt.

...

31.6 Übersicht

31.7 Rückgabetypen

31.8 Casten

// Bruchklasse

Kapitel 32

Klassen als Datenelemente einer Klasse

Analog zu Basistyp-Variablen, können Sie auch Klassenobjekte als Member einer Klasse deklarieren. Dies funktioniert genau so, wie Sie es schon kennen.

```
class A{
    int zahl;
};
class B{
    A a_objekt;
};
```

Nun enthält die Klasse `B` ein Objekt der Klasse `A`.

32.1 Verweise untereinander

Manchmal ist es nötig, das zwei Klassen einen Verweis auf die jeweils andere enthalten. In diesem Fall muss die Klassendeklaration von der Klassendefinition getrennt werden. Wie auch bei Funktionen gilt, dass jede Definition immer auch eine Deklaration ist und das Deklaration beliebig oft gemacht werden können, während Definition nur einmal erfolgen dürfen.

```
// Deklaration von A
```

```
class A;
// Definition von A
class A{
    int zahl;
};
```

Um ein Klassenobjekt zu Deklarieren, muss die gesamte Klassendefinition bekannt sein. Wenn Sie hingegen nur einen Zeiger oder eine Referenz auf ein Klassenobjekt deklarieren möchten, genügt es wenn die Klasse deklariert wurde.

```
class A; // Deklaration von A
class B{
    // B enthält eine Referenz auf ein A-Objekt
    A& a_objekt;
};
class A{
    // A enthält ein B-Objekt
    B objekt;
};
```

Falls Ihnen jetzt die Frage im Hinterkopf herumschwirrt, ob es möglich ist, das beide Klassen ein Objekt der jeweils anderen enthalten, dann denken Sie noch mal darüber nach was diese Aussage bedeuten würde. Die Antwort auf diese Frage ist selbstverständlich nein.

Kapitel 33

Rechnen mit Brüchen

Das Rechnen mit Ganzzahlen und Gleitkommazahlen haben Sie inzwischen gelernt. In diesem Kapitel kehren wir das Prinzip mal um. Jetzt werden Sie Ihrem Rechner beibringen mit Brüchen zu rechnen! Davon ausgehend das Sie wissen was Brüche sind, sollte das kein all zu großes Problem darstellen. Die Klasse die am Ende dieses Abschnittes entstanden ist wird noch nicht perfekt sein, aber Sie wird zum normalen Rechnen ausreichen. Sie dürfen sie natürlich jederzeit weiterentwickeln und verbessern.

33.1 Was diese Klasse bieten soll

Bevor wir nun Hals über Kopf anfangen Quellcode zu schreiben, sollten wir uns erst klar machen, was genau wir überhaupt erreichen wollen.

Brüche bestehen aus einem Zähler und einem Nenner. Diese sind Ganzzahlen, allerdings ist der Nenner eine Natürliche Zahl (ohne Vorzeichen) und der Zähler eine Ganze Zahl (mit Vorzeichen). Daher werden wir den entsprechenden Variablen passende Datentypen zuordnen.

Da es beim Rechnen leichter ist, die Brüche gleich in ihrer kleinsten (nicht mehr zu kürzenden) Form zu sehen, werden wir den Bruch nach jeder Rechenoperation kürzen. Weiterhin wollen wir mit den Brüchen rechnen können wie mit den eingebauten Datentypen für Zahlen. Natürlich brauchen wir eine Ein- und Ausgabe für unsere Brüche. Schließlich und endlich wollen wir Sie beim Rechnen mit den eingebauten Datentypen mischen.

33.2 Was diese Klasse nicht kann

Die logische Folge des automatischen Kürzens ist natürlich, dass sich die Brüche nicht mit beliebigen Zahlen erweitern und kürzen lassen. Es ist natürlich möglich, beides gleichzeitig zu realisieren, aber es ist nicht sinnvoll, also machen wir das auch nicht. Ein Umwandlung von Gleitkommazahlen in gebrochene Zahlen ist ebenfalls nicht vorgesehen.

33.3 Ein erster Blick

Gleich werden Sie zum ersten mal einen Blick auf die Bruch-Klasse werfen, also machen Sie sich bereit:

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);
private:
    int          m_zaehler;
    unsigned int m_nenner;
};
```

Das war nicht besonders aufregend, die Klasse hat bisher nur einen Konstruktor und die Variablen `m_zaehler` und `m_nenner`. Beachten Sie aber das `m_zaehler` vom Datentyp `int` (Vorzeichenbehaftet) ist, während `m_nenner` den Datentyp `unsigned int` (Vorzeichenlos) hat.

Auf die Standardparameter des Konstruktors werden wir in einem späteren Kapitel noch einmal zu sprechen kommen. Für den Moment soll es reichen, seine Deklaration zu zeigen.

Kapitel 34

Die Methoden

In diesem Kapitel werden Sie mit den Methoden bekannt gemacht die unsere Bruch-Klasse zur Verfügung stellt. Grundlegend tauchen im Zusammenhang mit Brüchen zwei Begriffe auf: „Erweitern“ und „Kürzen“. Im Zusammenhang mit diesen beiden finden sich wiederum die Begriffe „kleinstes gemeinsames Vielfaches“ und „größter gemeinsamer Teiler“. Wir haben uns bei dem Vorüberlegungen dafür entschieden das Erweitern wegzulassen, beim der Addition und Subtraktion brauchen wir es aber oder genauer gesagt wir brauchen das damit in Zusammenhang stehende „kleinste gemeinsame Vielfache“. Im folgenden werden „kleinste gemeinsame Vielfache“ und „größter gemeinsamer Teiler“ mit kgV und ggT abgekürzt.

Wir haben also die folgenden Methoden:

- ggT
- kgV
- kuerzen

Dazu kommen noch 2 Zugriffsfunktionen, da sich der Benutzer unserer Klasse vielleicht für Zähler und Nenner des Bruchs interessiert:

- zaehler
- nenner

2 dieser 5 Methoden müssen aber gleich wieder gestrichen werden. ggT und kgV sind zwar in Zusammenhang mit Brüchen hilfreich, aber sie beschreiben allgemeine Mathematische Funktionen die nicht ausschließlich in Zusammenhang mit Brüchen eingesetzt werden. Daher sind diese beiden *Funktionen* unter der über-

schrift Methoden etwas fehlpositioniert, da Sie keine Member der Klasse Bruch sind.

34.1 Zugriff

Die beiden Zugriffsmethoden tun nichts weiter als Zähler und Nenner des Bruchs zurückzugeben, daher schreiben wir Sie direkt in die Klassendeklaration.

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);
    int          zaehler()const {return m_zaehler;}
    unsigned int nenner()const  {return m_nenner;}
private:
    int          m_zaehler;
    unsigned int m_nenner;
};
```

Wie Sie sehen, haben die beiden Methoden den gleiche Rückgabetyp wie die Variablen, die Sie repräsentieren. Achten Sie bitte auch darauf, dass beide als `const` Deklariert sind, da Sie keine Variable innerhalb der Klasse verändern.

34.2 ggT()

In der Schule haben Sie sicher gelernt, dass sich der ggT (größter gemeinsamer Teiler) durch Primfaktorzerlegung ermitteln lässt. Dieses Verfahren ist allerdings denkbar ungünstig, um es auf einem Rechner umzusetzen. Sie müssten zunächst mal die Primzahlen berechnen und das dauert, zumal Sie ja gar nicht wissen wie viele Primzahlen Sie überhaupt benötigen.

Deshalb bedingen wir uns eines anderen Verfahrens, dem Euklidischen Algorithmus. Eine Verbesserung dieses Verfahrens ist der Steinsche Algorithmus, aber für unsere Zwecke reicht ersterer. Sie dürfen die Klasse natürlich gerne dahingehend verbessern, das Sie den Steinschen Algorithmus einsetzen.

Der euklidische Algorithmus beruht auf zwei Eigenschaften des größten gemeinsamen Teilers:

$$\text{ggT}(q \cdot b + r, b) = \text{ggT}(b, r)$$

und

$$\text{ggT}(a, 0) = a$$

Wenn Sie eine genaue Beschreibung wünschen, dann schauen Sie sich doch mal den entsprechenden Wikipediaartikel an.

```
unsigned int ggT(unsigned int a, unsigned int b){
    if(b == 0)
        return a;
    else return ggT(b, a % b);
}
```

Beachten Sie, dass diese Funktion rekursiv funktioniert. Die Implementierung als einfache Schleife, wäre ebenfalls möglich gewesen, aber sie ist etwas unübersichtlicher und dank Optimierung, ist die rekursive Variante nur unwesentlich langsamer. Wenn Sie nicht mehr wissen, was rekursive Funktionen sind, dann werfen Sie noch mal einen Blick auf das Kapitel „Schleifen mal anders – Rekursion“ im Abschnitt „Weitere Grundelemente“.

34.3 kgV()

Das kgV lässt sich ganz einfach Berechnen wenn Sie das ggT kennen. Daher schreiben wir die Funktion mit Hilfe der eben erstellten ggT-Funktion.

```
unsigned int kgV(unsigned int a, unsigned int b){
    return a/ggT(a,b) * b;
}
```

Hinweis
Damit kgV() auf ggT zugreifen kann, muss ggT() natürlich bekannt sein. Sie müssen die Deklaration (in diesem Fall den Prototyp von ggT() immer geschrieben haben, bevor Sie das entsprechende Element verwenden.

34.4 kuerzen()

kuerzen() ist nun endlich mal wirklich eine Memberfunktion (oder auch Methode) von Bruch. Da sie somit direkten Zugriff auf die Variablen hat die sie verändern soll, braucht sie keine Argumente und auch keinen Rückgabewert.

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);
    int          zaehler()const {return m_zaehler;}
    unsigned int nenner()const  {return m_nenner;}
private:
    void kuerzen();
    int          m_zaehler;
    unsigned int m_nenner;
};
void Bruch::kuerzen(){
    unsigned int tmp(ggT(m_zaehler, m_nenner));
    m_zaehler /= tmp;
    m_nenner  /= tmp;
}
```

Da kuerzen() eine Methode ist, die nur innerhalb der Klasse nach einer Rechenoperation aufgerufen wird, deklarieren wir sie im privaten Bereich der Klasse.

Kapitel 35

Die Rechenoperationen

Addition, Subtraktion, Multiplikation und Division, das sind 4 Rechenoperationen. C++ bietet aber die mit der Zuweisung kombinierten Kurzschreibweisen, womit wir insgesamt auf 8 kommen.

35.1 Addition

Um zwei Brüche zu Addieren, müssen die Nenner gleich sein. Wenn wir bereits Brüche haben die sich nicht weiter kürzen lassen, (und dafür sorgt unsere Klasse,) dann erhalten wir wiederum einen unkürzbaren Bruch, wenn wir mit dem kgV erweitern. Das ganze sieht also so aus:

```
ErgebnisNenner = kgV(Bruch1Nenner, Bruch2Nenner);
ErgebnisZähler = Bruch1Zähler * (ErgebnisNenner / Bruch2Nenner) +
    Bruch2Zähler * (ErgebnisNenner / Bruch1Nenner);
```

35.2 Subtraktion

Für die Subtraktion gelten die gleichen Regeln wie bei der Addition.

```
ErgebnisNenner = kgV(Bruch1Nenner, Bruch2Nenner);
ErgebnisZähler = Bruch1Zähler * (ErgebnisNenner / Bruch2Nenner) -
```

```
    Bruch2Zähler * (ErgebnisNenner / Bruch1Nenner);
```

35.3 Multiplikation

Bei der Multiplikation werden einfach die Zähler und die Nenner multipliziert. Danach muss der Bruch wieder gekürzt werden.

```
ErgebnisZähler = Bruch1Zähler * Bruch2Zähler;
ErgebnisNenner = Bruch1Nenner * Bruch2Nenner;
kuerzen()
```

35.4 Division

Die Division stimmt mit der Multiplikation fast überein, aber statt die Zähler und die Nenner miteinander zu multiplizieren, werden Sie gegeneinander multipliziert.

```
ErgebnisZähler = Bruch1Zähler * Bruch2Nenner;
ErgebnisNenner = Bruch1Nenner * Bruch2Zähler;
kuerzen()
```

35.5 Kombination

Da C++ wie schon gesagt neben den normalen Rechenoperatoren noch die mit der Zuweisung kombinierten zu Verfügung stellt, werden wir einen kleinen Trick anwenden, um uns doppelte Arbeit zu ersparen. Wir werden die eigentlichen Rechenoperationen in den Zuweisungskombioperatoren implementieren und dann innerhalb der normalen Rechenoperatoren temporäre Objekte anlegen, für welche wir die Kombinationsoperatoren aufrufen. Das ist ein übliches und vielangewantes Verfahren, welches einige Vorteile zu bieten hat. Sie sparen doppelte Schreibarbeit und müssen sich bei Veränderungen nur um die Kombioperatoren kümmern, da sich die anderen ja genauso verhalten.

Die umgekehrte Variante, also von den Kombioperatoren die normalen aufrufen zu lassen, ist übrigens nicht zu empfehlen, da die Kombinationsoperatoren immer schneller sind, sie benötigen schließlich keine temporären Objekte. Außerdem ist es in vielen Klassen nötig, die normalen Rechenoperatoren außerhalb der Klasse zu deklarieren. Wenn Sie nicht als `friend` deklariert sind, haben Sie keinen Zugriff auf die privaten Member der Klasse, rufen Sie dagegen die Kombioperatoren auf, brauchen Sie gar keinen Zugriff.

35.6 Abschluss

So, nun haben Sie wirklich genug Theorie gehört, es wird Zeit zu zeigen wie das ganze im Quelltext aussieht. Der Code lässt sich zwar noch nicht ausführen, weil der Konstruktor noch nicht definiert ist, aber es lohnt sich trotzdem schon mal einen Blick darauf zu werfen.

```
unsigned int ggT(unsigned int a, unsigned int b){
    if(b == 0)                 // Wenn b gleich 0
        return a;              // ggT gefunden
    else return ggT(b, a % b); // andernfalls weitersuchen
}
unsigned int kgV(unsigned int a, unsigned int b){
    // Das kgV zweier Zahlen, ist ihr Produkt geteielt durch ihren ggT
    return a * b / ggT(a, b);
}
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1); // noch nicht definiert
    int          zaehler()const {return m_zaehler;}  // Gibt Zähler zurück
    unsigned int nenner()const  {return m_nenner;}   // Gibt Nenner zurück
    Bruch& operator+=(Bruch const &lvalue);
    Bruch& operator-=(Bruch const &lvalue);
    Bruch& operator*=(Bruch const &lvalue);
    Bruch& operator/=(Bruch const &lvalue);
    // Diese Methoden erstellen eine Temporäre Kopie ihres Objekts, führen
    // die Rechenoperation auf ihr aus und geben sie dann zurück
    Bruch operator+(Bruch const &lvalue)const{return Bruch(*this)+=lvalue;}
    Bruch operator-(Bruch const &lvalue)const{return Bruch(*this)-=lvalue;}
    Bruch operator*(Bruch const &lvalue)const{return Bruch(*this)*=lvalue;}
```

```
    Bruch operator/(Bruch const &lvalue)const{return Bruch(*this)/=lvalue;}
private:
    void kuerzen();                                 // kürzt weitestmöglich
    int          m_zaehler;
    unsigned int m_nenner;
};
void Bruch::kuerzen(){
    const unsigned int tmp = ggT(m_zaehler, m_nenner);     // ggT in tmp speichern
    m_zaehler /= tmp; // Zähler durch ggT teilen
    m_nenner  /= tmp; // Nenner durch ggT teilen
}
Bruch& Bruch::operator+=(Bruch const &lvalue){
    const unsigned int tmp = kgV(m_nenner, lvalue.m_nenner);
    m_zaehler = m_zaehler * (tmp / m_nenner) + lvalue.m_zaehler * (tmp / lvalue.m_nenner);
    m_nenner  = tmp;
    return *this; // Referenz auf sich selbst zurückgeben
}
Bruch& Bruch::operator-=(Bruch const &lvalue){
    const unsigned int tmp = kgV(m_nenner, lvalue.m_nenner);
    m_zaehler = m_zaehler * (tmp / m_nenner) - lvalue.m_zaehler * (tmp / lvalue.m_nenner);
    m_nenner  = tmp;
    return *this; // Referenz auf sich selbst zurückgeben
}
Bruch& Bruch::operator*=(Bruch const &lvalue){
    m_zaehler *= lvalue.m_zaehler;
    m_nenner  *= lvalue.m_nenner;
    kuerzen();    // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}
Bruch& Bruch::operator/=(Bruch const &lvalue){
    m_zaehler *= lvalue.m_nenner;
    m_nenner  *= lvalue.m_zaehler;
    kuerzen();    // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}
```

Kapitel 36

Umwandlung aus anderen Datentypen

Um aus einer Variable eines andren Datentyps in einen Bruch umzuwandeln, übergibt man einem Konstruktor diese Variable und lässt ihn die Umwandlung durchführen. Sie erinnern sich bestimmt noch daran, dass im ersten Kapitel dieses Abschnittes stand, die Standardparameter des Konstruktors würden später besprochen. Dieser Zeitpunkt ist nun gekommen. Zur Erinnerung, seine Deklaration innerhalb der Klasse lautete:

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);
// ...
};
Bruch::Bruch(int zaehler, unsigned int nenner):
    m_zaehler(zaehler),
    m_nenner(nenner){
    kuerzen();
}
```

Die Definition steht, wie Sie oben sehen, außerhalb der Klasse. Innerhalb der Initialisierungsliste, welche durch einen Doppelpunkt eingeleitet wird, werden die Membervariablen mit den übergebenen Parametern initialisiert. Innerhalb des Funktionsrumpfes wird die Methode `kuerzen()` aufgerufen, um den Buch falls nötig zu kürzen.

Beachten Sie, dass die Standardparameter nur bei der Deklaration, nicht aber bei der Definition einer Funktion angegeben werden. Unser Konstruktor übernimmt 2 `int`-Werte und beide besitzen einen Standardwert, daher kann er in 3 Formen aufgerufen werden:

```
Bruch bruch1()     // erzeugt (0/1)
Bruch bruch2(5)    // erzeugt (5/1)
Bruch bruch3(3, 4) // erzeugt (3/4)
```

Die erste Form entspricht einem Defaultkonstruktor und setzt den Wert des Buches auf `(0/1)`, was dem ganzzahligen Wert `0` entspricht. In der zweiten Form wird 1 `int`-Wert übergeben, der resultierende Buch entspricht diesem Wert, da der Nenner dank des Standardparameters auf 1 gesetzt wird. Die dritte Form übernimmt schließlich 2 `int`-Werte, der erste gibt den Zähler und der zweite den Nenner des Bruches an.

Auch an dieser Stelle soll noch einmal darauf hingewiesen werden, dass der Nenner nicht negativ sein kann, daher ist dieser Parameter auch vom Typ `unsigned int` welcher nur positive Werte zulässt.

Thema wird Später näher erläutert...
Bitte beachten Sie, dass der Nenner eines Buches normalerweise nicht 0 sein kann. Normalerweise würde man dem in C++ mit einer Ausnahme begegnen. Da wir Ausnahmen aber bisher nicht besprochen haben und es auch noch eine Weile dauern wird, bis wir dieses Thema behandeln, werden wir dem Umstand, dass es möglich ist dem Konstruktor von Buch eine 0 als Nenner zu übergeben, erst einmal ignorieren. Später kommen wir auf diesen Punkt aber noch einmal zurück.

36.1 Gleitkommazahl wird Bruch

Wir haben jetzt die Umwandlung von ganzen Zahlen in Brüche besprochen. Als nächstes wollen wir eine Gleitkommazahl in einen Bruch umwandeln. Hierfür benötigen wir einen zweiten Konstruktor, welcher eine Gleitkommazahl übernimmt und Sie in einen Bruch umwandelt:

```
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1);
```

```
    Bruch(double wert);
// ...
};
Bruch::Bruch(double wert):
    m_zaehler(static_cast<int>(wert*1000000.0+0.5)),
    m_nenner(1000000){
    kuerzen();
}
```

Es ist kaum möglich, einen `double`-Wert zuverlässig in einen Bruch umzuwandeln, da die Datentypen, die wir für Zäher (`int`) und Nenner (`unsigned int`) verwenden, den Wertebereich unserer Brüche gewaltig einschränken. Dieser Konstruktor ist daher kaum praxistauglich, aber als Beispiel sollte er genügen. Einmal vorausgesetzt, dass ein `int` 4 Byte groß ist, beachtet er 3 Vorkomma- und 6 Nachkommastellen.

Wir setzen den Nenner des Buches auf `1000000`, dann multiplizieren wir den übergebenen Wert mit 1000000. Würde man jetzt Zähler durch Nenner teilen, hätte man wieder exakt den Wert, der übergeben wurde. Da der Zähler aber ein `int`- und kein `double`-Wert sein muss, müssen wir ihn noch umwandeln. Da eine solche Umwandlung aber alle Nachkommastellen abschneidet, anstatt kaufmännisch korrekt zu runden, addieren wir den `double`-Wert vorher mit `0.5`, was dazu führt, das die Zahl nach den abschneiden der Nachkommastellen kaufmännisch gerundet wurde.

Alles was über 6 Nachkommastelle hinausgeht, ist also für uns nicht relevant, da es korrekt gerundet wird. Das Problem besteht darin, dass der Bruch keinen Zahler aufnehmen kann, der größer als der größtmögliche `int`-Wert (bei 4 Byte `2147483647`) ist. Leider lässt sich dieses Problem nicht ohne weiteres lösen, daher werden wir damit leben, das dies nur ein Beispiel für die Umwandlung einer Gleitkommazahl in einen Bruch ist und keine perfekte Implementierung.

Anschließend wir im Funktionsrumpf wieder einmal die Funktion `kuerzen()` aufgerufen. Sie können nun also auch schreiben:

```
Bruch bruch1(0.25)       // erzeugt (1/4)
Bruch bruch2(7.0)        // erzeugt (7/1)
Bruch bruch3(999.999999) // erzeugt (999999999/1000000)
```

36.2 (k)ein Kopierkonstruktor

Für unsere Bruch-Klasse werden wir keinen Kopierkonstruktor anlegen, da unser Compiler uns diese Arbeit mit einem zufriedenstellenden Ergebnis abnimmt. Oder anders ausgedrückt, Sie können ohne eine einzige Zeile Code zur Klasse hinzuzufügen schreiben:

```
Bruch bruch1(1, 5);   // erzeugt (1/5)
Bruch bruch2(bruch1); // erzeugt (1/5)
```

Kapitel 37

Ein- und Ausgabe

Als nächstes wollen wir dafür sorgen, das unsere Brüche ebenso einfach ein- und ausgegeben werden können, wie die elementaren Datentypen. Sie wissen ja bereits, dass Sie hierfür nur den Ein- und Ausgabeoperator überladen müssen. Wir wollen die Brüche in der folgenden Form schreiben und lesen: `(Zähler/Nenner)`

Die beiden folgenden Operatorfunktionen können Sie nach der Deklaration der `Bruch`-Klasse einfügen. Includedateien gehören natürlich an die übliche Stelle am Dateianfang.

37.1 Ausgabe

Die Ausgabe lässt sich ziemlich simpel realisieren, wir geben einfach das von uns gewünschte Format auf dem Ausgabestream aus, den wir mittels Parameter erhalten. Dann geben wir den Stream zurück. Beachten Sie allerdings, dass Sie die Headerdatei „ios“ includieren müssen, damit Sie die Standardstreams überladen können.

```
   1. include <ios>


std::ostream& operator<<(std::ostream &os, Bruch const &bruch){
    return os << '(' << bruch.zaehler() << '/' << bruch.nenner() << ')';
}
```

37.2 Eingabe

Die Eingabe ist etwas schwieriger. Wir müssen sicherstellen, dass das von uns vorgegebene Format eingehalten wird und falls nötig, den Eingabestream auf einen Fehlerstatus zu setzten.

```
    1. include <ios>


std::istream& operator>>(std::istream &is, Bruch &bruch){
    char tmp;
    int zaehler, nenner;
    is >> tmp;
    if(tmp=='('){
        is >> zaehler;
        is >> tmp;
        if(tmp=='/'){
            is >> nenner;
            is >> tmp;
            if(tmp==')'){
                bruch=Bruch(zaehler, nenner);
                return is;
            }
        }
    }
    is.setstate(std::ios_base::failbit);
    return is;
}
```

Wie Sie sehen können, wird diesmal eine nichtkonstante Referenz auf einen Bruch übergeben, da dieser ja geändert wird, wenn man Ihm einen neuen Wert zuweist. Da die Klasse `Bruch` keine Möglichkeit bereitstellt, `zaehler` oder `nenner` einzeln zu ändern, erstellen wir nach dem Einlesen beider einen entsprechenden `Bruch` und weisen Ihn an unseren zu. Auf diese Weise stellen wir auch sicher, dass sich an dem Bruch nichts verändert, wenn während des Einlesens irgendetwas schiefgeht. Falls etwas daneben geht, wird das `failbit` gesetzt und anschließend der Stream zurückgegeben.

Kapitel 38

Umwandlung in andere Datentypen

In diesem Kapitel werden wir unseren Bruch in Gleitkommazahlen umwandeln. Hierfür müssen wir einfach den Zähler durch den Nenner teilen. Da jedoch beide einen integralen Typ haben, müssen wir vor dem Teilen einen der Werte in eine Gleitkommazahl umwandeln, damit keine Ganzzahldivision durchgeführt wird. Bei einer Ganzzahldivision würden natürlich die Nachkommastellen abgeschnitten.

```
class Bruch{
public:
// ...
    operator float()      {return static_cast<float>(m_zaehler) / m_nenner;}
    operator double()     {return static_cast<double>(m_zaehler) / m_nenner;}
    operator long double(){return static_cast<long double>(m_zaehler) / m_nenner;}
// ...
};
```

Sie können den Bruch jetzt auf die folgende Weise in eine Gleitkommazahl umwandeln:

```
    1. include <iostream>


// Alles was zur Bruch-Klasse gehört
int main(){
    std::cout << static_cast<double>(Bruch(1, 8));
}
```

Ausgabe

```
0.125
```

Kapitel 39

Der Taschenrechner geht zur Schule

Im Abschnitt „Einführung in C++“ gab es ein Kapitel über einen einfachen Taschenrechner. In diesem Kapitel werden wir ihn ganz leicht modifizieren, so das er mit Brüchen statt mit Gleitkommazahlen rechnet und die Ausgabe als Brüche und als Gleitkommazahlen gemacht wird.

```
    1. include <iostream>


// Alles was zur Bruch-Klasse gehört
int main(){
    Bruch zahl1, zahl2, ergebnis;                    // Variablen für Zahlen vom Typ Bruch
    char rechenzeichen;                              // Variable fürs Rechenzeichen

    std::cout << "Geben Sie eine Rechenaufgabe ein: "; // Eingabeaufforderung ausgeben
    std::cin >> zahl1 >> rechenzeichen >> zahl2;     // Aufgabe einlesen

    switch(rechenzeichen){                           // Wert von rechenzeichen ermitteln
        case '+': ergebnis = zahl1+zahl2; break;     // entsprechend dem
        case '-': ergebnis = zahl1-zahl2; break;     // Rechenzeichen
        case '*': ergebnis = zahl1*zahl2; break;     // das Ergebnis
        case '/': ergebnis = zahl1/zahl2; break;     // berechnen
        // Fehlerausgabe und Programm beenden, falls falsches Rechenzeichen
        default: std::cout << "unbekanntes Rechenzeichen...\n"; return 1;
    }

    // Aufgabe noch mal komplett ausgeben
    std::cout << zahl1 << ' ' << rechenzeichen << ' ' << zahl2 << " = " << ergebnis << '\n';
    std::cout << static_cast<double>(zahl1) << ' '   // Ausgabe als
```

```
            << rechenzeichen << ' '                  // Gleitkommawerte
            << static_cast<double>(zahl2) << " = "
            << static_cast<double>(ergebnis) << '\n';
}
```

Ausgabe

```
Geben Sie eine Rechenaufgabe ein: <Eingabe>(1/4)*(1/2)</Eingabe>
(1/4) * (1/2) = (1/8)
0.25 * 0.5 = 0.125
```

In der Zusammenfassung finden Sie noch einmal das gesamte Programm.

Kapitel 40

Zusammenfassung

Hier finden Sie noch einmal das komplette Programm, inklusive der Ausgabe eines Durchlaufs.

```
    1. include <iostream> // Bindet auch die Datei <ios> ein


unsigned int ggT(unsigned int a, unsigned int b){
    if(b == 0)
        return a;
    else return ggT(b, a % b);
}
unsigned int kgV(unsigned int a, unsigned int b){
    return a/ggT(a,b) * b;
}
class Bruch{
public:
    Bruch(int zaehler = 0, unsigned int nenner = 1); // Konstruktoren
    Bruch(double wert);                              // dieser ist nicht perfekt
    int          zaehler()const {return m_zaehler;}  // Gibt Zähler zurück
    unsigned int nenner()const  {return m_nenner;}   // Gibt Nenner zurück
    Bruch& operator+=(Bruch const &lvalue);
    Bruch& operator-=(Bruch const &lvalue);
    Bruch& operator*=(Bruch const &lvalue);
    Bruch& operator/=(Bruch const &lvalue);
    // Diese Methoden erstellen eine Temporäre Kopie ihres Objekts, führen
    // die Rechenoperation auf ihr aus und geben sie dann zurück
    Bruch operator+(Bruch const &lvalue)const{return Bruch(*this)+=lvalue;}
    Bruch operator-(Bruch const &lvalue)const{return Bruch(*this)-=lvalue;}
```

```
    Bruch operator*(Bruch const &lvalue)const{return Bruch(*this)*=lvalue;}
    Bruch operator/(Bruch const &lvalue)const{return Bruch(*this)/=lvalue;}
    // Umwandlung in Gleitkommatypen
    operator float()      {return static_cast<float>(m_zaehler)/m_nenner;}
    operator double()     {return static_cast<double>(m_zaehler)/m_nenner;}
    operator long double(){return static_cast<long double>(m_zaehler)/m_nenner;}
private:
    void kuerzen();                                  // kürzt weitestmöglich
    int          m_zaehler;
    unsigned int m_nenner;
};
Bruch::Bruch(int zaehler, unsigned int nenner):
    m_zaehler(zaehler),
    m_nenner(nenner){
    kuerzen();
}
Bruch::Bruch(double wert):
    m_zaehler(static_cast<int>(wert*1000000.0+0.5)),
    m_nenner(1000000){
    kuerzen();
}
void Bruch::kuerzen(){
    unsigned int tmp = ggT(m_zaehler, m_nenner); // ggT in tmp speichern
    m_zaehler /= tmp;                            // Zähler durch ggT teilen
    m_nenner  /= tmp;                            // Nenner durch ggT teilen
}
Bruch& Bruch::operator+=(Bruch const &lvalue){
    unsigned int tmp = kgV(m_nenner, lvalue.m_nenner);
    m_zaehler = m_zaehler * (tmp / m_nenner) + lvalue.m_zaehler * (tmp / lvalue.m_nenner);
    m_nenner  = tmp;
    return *this; // Referenz auf sich selbst zurückgeben
}
Bruch& Bruch::operator-=(Bruch const &lvalue){
    unsigned int tmp = kgV(m_nenner, lvalue.m_nenner);
    m_zaehler = m_zaehler * (tmp / m_nenner) - lvalue.m_zaehler * (tmp / lvalue.m_nenner);
    m_nenner  = tmp;
    return *this; // Referenz auf sich selbst zurückgeben
}
Bruch& Bruch::operator*=(Bruch const &lvalue){
    m_zaehler *= lvalue.m_zaehler;
    m_nenner  *= lvalue.m_nenner;
```

```
    kuerzen();    // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}
Bruch& Bruch::operator/=(Bruch const &lvalue){
    m_zaehler *= lvalue.m_nenner;
    m_nenner  *= lvalue.m_zaehler;
    kuerzen();    // Bruch wieder kürzen
    return *this; // Referenz auf sich selbst zurückgeben
}
std::ostream& operator<<(std::ostream &os, Bruch const &bruch){
    return os << '(' << bruch.zaehler() << '/' << bruch.nenner() << ')';
}
std::istream& operator>>(std::istream &is, Bruch &bruch){
    char tmp;
    int zaehler, nenner;
    is >> tmp;
    if(tmp=='('){
        is >> zaehler;
        is >> tmp;
        if(tmp=='/'){
            is >> nenner;
            is >> tmp;
            if(tmp==')'){
                bruch=Bruch(zaehler, nenner); // Bruch erzeugen und Wert übernehmen
                return is;
            }
        }
    }
    is.setstate(std::ios_base::failbit);      // Fehlerstatus setzen
    return is;
}
int main(){
    Bruch zahl1, zahl2, ergebnis;                     // Variablen für Zahlen vom Typ Bruch
    char rechenzeichen;                               // Variable fürs Rechenzeichen
    std::cout << "Geben Sie eine Rechenaufgabe ein: "; // Eingabeaufforderung ausgeben
    std::cin >> zahl1 >> rechenzeichen >> zahl2;      // Aufgabe einlesen
    switch(rechenzeichen){                            // Wert von rechenzeichen ermitteln
        case '+': ergebnis = zahl1+zahl2; break;      // entsprechend dem
        case '-': ergebnis = zahl1-zahl2; break;      // Rechenzeichen
        case '*': ergebnis = zahl1*zahl2; break;      // das Ergebnis
        case '/': ergebnis = zahl1/zahl2; break;      // berechnen
```

```
        // Fehlerausgabe und Programm beenden, falls falsches Rechenzeichen
        default: std::cout << "unbekanntes Rechenzeichen...\n"; return 1;
    }
    // Aufgabe noch mal komplett ausgeben
    std::cout << zahl1 << ' ' << rechenzeichen << ' ' << zahl2 << " = " << ergebnis << '\n';
    std::cout << static_cast<double>(zahl1) << ' '     // Ausgabe als
              << rechenzeichen << ' '                  // Gleitkommawerte
              << static_cast<double>(zahl2) << " = "
              << static_cast<double>(ergebnis) << '\n';
}
```

Ausgabe

```
Geben Sie eine Rechenaufgabe ein: <Eingabe>(1/5)-(3/4)</Eingabe>
(1/5) - (3/4) = (-11/20)
0.2 - 0.75 = -0.55
```

Kapitel 41

Nochmal Klassen

Neben den Klassen gibt es in C++ auch noch die aus C stammenden Strukturen. Eine Struktur, auch Datenverbund genannt, fasst mehrere Variablen zu einem Typ zusammen. Im Gegensatz zu den C-Strukturen sind C++-Strukturen allerdings vollständig kompatibel zu Klassen. Sie können in einer Struktur also problemlos Methoden deklarieren oder die Zugriffsrechte über die Schlüsselworte `public`, `protected` (wird später Erläutert) und `private` festlegen.

Der einzige wirkliche Unterschied zwischen Strukturen und Klassen ist in C++, das Klassen implizit `private`, während Strukturen implizit `public` sind. Da dieser impliziten Zugriffsrechtebereich aber für gewöhnlich nicht genutzt werden sollte, ist dieser Unterschied fast unbedeutend. Das folgende kleine Beispiel zeigt die Definition einer Struktur im vergleich zu der einer Klasse.

```
struct A{
    // öffentliche Member
public:
    // öffentliche Member
private:
    // private Member
};
class B{
    // private Member
public:
    // öffentliche Member
private:
    // private Member
```

```
};
```

Thema wird Später näher erläutert...
Im Kapitel über Vererbung werden Sie noch sehen, das Sie eine Klasse problemlos von einer Struktur ableiten können und umgekehrt.

Kapitel 42

Wir empfehlen inline

In den Kapiteln über Funktionen und Headerdateien haben Sie bereits einiges über `inline`-Funktionen erfahren. An dieser Stelle soll dieses Wissen nun auf den Gebrauch von `inline`-Klassenmethoden übertragen werden. Das folgende kleine Bespiel zeigt einer Klasse mit einer `inline`-Methode.

```
class A{
public:
    int methode()const;
private:
    int m_a;
};
inline int A::methode()const{
    return m_a;
}
```

Das Schlüsselwort `inline` steht vor der Methodendefinition, in der Deklaration würde es zwar keinen Compilierfehler produzieren, aber es würde auch keinerlei Auswirkungen haben. Ein `inline` vor einer Methodendeklaration kann genutzt werden um andere Programmierer darauf hinzuweisen, das diese Funktion `inline` sein soll, für den Compiler währe es bedeutungslos. Damit der Compiler den Methodenaufruf auch durch den Methodenrumpf ersetzen kann, steht die Methodendefinition ebenso die Klassendefinition in der Headerdatei und nicht in einer eventuell zugehörigen Implementierungsdatei.

Wenn Sie eine Funktion direkt innerhalb einer Klassendefinition definieren, ist Sie implizit `inline`, da solche Funktion nahezu immer sehr kurz sind und sich somit gut zum inlinen eignen.

```
class A{
public:
    // methode ist implizit inline
    int methode()const { return m_a; }
private:
    int m_a;
};
```

Kapitel 43

Vererbung

43.1 Einleitung

Wenn von Objektorientierung gesprochen wird fällt früher oder später auch das Stichwort Vererbung. Auf dieser Seite lernen Sie anhand eines Beispieles die Grundprinzipien der Vererbung kennen.

43.2 Die Ausgangslage

Stellen wir uns vor, dass wir in einer Firma arbeiten und in einem Programm sämtliche Personen verwalten die mit dieser Firma in einer Beziehung stehen. Über jede Person sind folgende Daten auf jeden Fall bekannt: Name, Adresse, Telefonnummer. Um alles zusammenzufassen haben wir uns eine Klasse geschrieben mit deren Hilfe wir eine einzelne Person verwalten können: (Aus Gründen der Einfachheit benutzen wir `string`s)

```
1. include<iostream>
2. include<string>


using namespace std;
class Person{
   public:
    Person(string Name, string Adresse, string Telefon) :m_name(Name), m_addr(Adresse),
 m_phon(Telfon){}
    string getName(){ return m_name; }
```

```
    string getAddr(){ return m_addr; }
    string getPhone(){ return m_phon; }
    void info(){ cout << "Name: " << m_name << " Adresse: " << m_addr << " Telfon: " << m_-
phone
 << endl; }
   private:
    string m_name;
    string m_addr;
    string m_phon;
};
```

Dies ist natürlich eine sehr minimale Klasse, aber schließlich geht es hier ja auch um die Vererbung :D.

43.3 Gemeinsam und doch getrennt

Die obige Klasse funktioniert ja eigentlich ganz gut, nur gibt es ein kleines Problem. In unserer Firma gibt es Mitarbeiter, Zulieferer, Kunden, Chefs, ... Diese sind zwar alle Personen sie haben jedoch jeweils noch zusätzliche Attribute (z. B. ein Mitarbeiter hat einen Lohn während ein Kunde eine KundenNr. hat). Jetzt könnten wir natürlich für jeden unterschiedlichen Typ eine eigene Klasse schreiben, den bereits vorhanden Code der Klasse `Person` hineinkopieren und schließlich die entsprechenden Erweiterungen vornehmen. Dieser Ansatz hat jedoch einige Probleme:

- Er ist unübersichtlich
- Es kann leicht zu Kopierfehlern kommen
- Soll `Person` geändert werden so muss jede Klasse einzeln bearbeitet werden.

Zum Glück bietet uns C++ aber ein mächtiges Hilfsmittel in Form der Vererbung. Anstatt alles zu kopieren können wir den Compiler anweisen die Klasse `Person` als Grundlage zu verwenden. Dies wird durch ein Beispiel klarer.

43.3.1 Beispiel

```
class Employee : public Person {
   public:
    Employee(string Name, string Adresse, string Telefon, int Gehalt, int MitNr):
            m_salary(Gehalt),
            m_number(MitNr)
            :Person(Name,Adresse,Telefon) {}
       int getSalary(){ return m_salary; }
    int getNumber(){ return m_number; }
   private:
    int m_salary;
    int m_number;
};
```

43.4 Erläuterung des Beispiels

Nun wollen wir uns der Analyse des Beispiels zuwenden um genau zu sehen was passiert ist.

Das wichtigste im Code ist ganz am Anfang: `class Employee : public Person`. Damit weisen wir den Compiler an alle Elemente aus `Person` auch in `Employee` zu übernehmen (z.B. hat `Employee` jetzt auch eine info Funktion und eine Membervariable `m_name`). Eine solche Klasse nennen wir abgeleitet/Kindklasse von `Person`.

Des weitern rufen wir im Konstruktor auch den Konstruktor von `Person` auf. Wir sparen uns also sogar diesen Code.

Benutzen können wir die Klasse wie gewohnt mit der Ausnahmen, das wir jetzt auch alle Methoden von `Person` aufrufen können:

```
Employee Mit("Erika Mustermann", "Heidestraße 17, Köln", "123/454", 4523, 12344209);
Mit.info();
cout << Mit.getSalary() << endl;
cout << Mit.getName() << endl;
```

Thema wird Später näher erläutert...
Warum wir das Schlüsselwort `public` verwenden wird im Kapitel „Private und geschützte Vererbung“ erklärt.

43.5 `protected`

Zum Schluss erweitern wir noch das Prinzip der Datenkapselung auf die Vererbung erweitern. Bis jetzt kennen wir die Schlüsselwörter `public` und `private`. Machen wir folgendes Experiment: Schreiben sie eine neue Membermethode von `Employee` und versuchen sie auf `m_name` aus der `Person`-Klasse zuzugreifen. Der Compiler wird einen Fehler ausgeben. Warum?

`m_name` wurde in der `Person`-Klasse als `private` deklariert, dass heißt es kann nur von Objekten dieser Klasse angesprochen werden, nicht aber von abgeleiteten Klassen wie z. B. `Employee`. Um zu vermeiden das wir `m_name` als `public` deklarieren müssen gibt es `protected`. Es verhält sich ähnlich wie `private`, mit dem Unterschied, dass auch Objekte von Kindklassen auf mit diesem Schlüsselwort versehene Member zugreifen können.

Kapitel 44

Methoden (nicht) überschreiben

44.1 Einleitung

Als Ausgangspunkt nehmen wir die beiden Klassen `Person` und `Employee` aus dem vorhergehenden Abschnitt.

44.2 2 gleiche Namen, 1 Aufruf

Wir haben der Klasse `Employee` zwar schon neue Funktionen hinzugefügt. Bis jetzt hatten diese einzigartige Namen, die nicht in der Basisklasse `Person` vorgekommen sind (z. B. `get_salary()`). Nun machen wir folgenden Versuch: Wir fügen `Employee` eine Methode `void info()` hinzu. Da ein solcher Member bereits in Person existiert erwarten wir, dass uns der Compiler einen Fehler ausgibt.

Entgegen unserer Erwartung jedoch wird alles fehlerfrei übersetzt. Dies kommt daher, dass die Methode aus der Elternklasse ganz einfach überschrieben wird (Dies ist aber nur der Fall, wenn die Signaturen der Methoden, also Name, Parameter und Rückgabewert übereinstimmen). Das heißt aber auch, dass wir aufpassen müssen, welches `void info()` wir aufrufen. Wenn wir nämlich in einer Memberfunktion von `Employee` `void info()` aufrufen, so wird nicht die ursprüngliche Funktion aus Person, sondern die „neue“ Implementierung aus `Employee` ausgeführt.

44.3 Die Methode des Vaters aufrufen

Um eine überschriebene Methode der Basisklasse aufzurufen benutzen wir folgenden Syntax:

```
«Klassenname»::«Methodenname»(«Parameter...»);
```

Hinweis
Obwohl es möglich ist jede Funktion einer Basisklasse zu überschreiben, ist dies nicht empfehlenswert. Besser ist die Benutzung von virtuelle Methoden die genau zu diesem Zweck vorhanden sind.

Kapitel 45

Private und geschützte Vererbung

45.1 Einleitung

Bis jetzt haben wir alle Vererbungen mit dem Schlüsselwort `public` vorgenommen (dies wird „öffentliche Vererbung“ genannt). Nun werden wir lernen was passiert wenn statt `public` `private` bzw. `protected` verwendet werden.

45.2 Private Vererbung

Verwenden wir private so bedeutet dies das alle Membervariablen bzw. Memberfunktion aus der Basisklasse private werden, von außen also nicht sichtbar sind.

45.3 Geschützte Vererbung

Geschütze Vererbung verläuft analog zur privaten Vererbung und sagt aus, dass alle Member der Elternklasse im Bereich `protected` stehen.

45.4 Wann wird was benutzt?

Um festzustellen wann welche Art von Vererbung eingesetzt wird gibt es zwei unterschiedliche Arten wie eine abgeleitete Klasse im Verhältnis zu ihrer Basisklasse stehen kann.

1. "ist ein" Kann man sagen "Klasse B ist eine Klasse A" so erbt Klasse B `public` von A. (Beispiel: Ein Arbeiter ist eine Person)
2. "hat ein" Kann man sagen "Klasse B hat ein Klasse A Object" so erbt Klasse B `private` von A. (Beispiel: Ein Mensch hat ein Herz)

Hinweis
Meistens wird mithilfe von `public` vererbt. Andere Typen von Vererbung werden nur selten bis gar nicht benutzt. Oft ist es sinnvoll statt einer privaten Vererbung ein Memberobjekt der entsprechenden Klasse zu verwenden

Kapitel 46

Funktionstemplates

Ein *Funktionstemplate* ist eine „Schablone“, die dem Compiler mitteilt, wie eine Funktion erzeugt werden soll. Aus einem Template können – semantisch gleichartige – Funktionen mit verschiedenen Parametertypen erzeugt werden. Für alle, die eben in Panik zu Wiktionary oder einem noch schwereren Wörterbuch gegriffen haben: Semantisch heißt hier so viel, wie allgemein gehalten in Bezug auf den Datentyp. Die Funktion hat also immer die gleich Parameteranzahl, die Parameter haben aber unterschiedliche Datentypen. Das ist zwar nicht ganz korrekt ausgedrückt, aber eine genaue Definition würde das Wort wieder so unverständlich machen, dass es ohne Erklärung weniger Schrecken verbreitet hätte. Lesen Sie einfach das Kapitel, dann werden Sie auf einige der „fehlenden Klauseln“ selbst aufmerksam werden.

Das nachfolgende Template kann Funktionen generieren, welche die größere von 2 Zahlen zurückliefern. Der Datentyp spielt dabei erst einmal keine Rolle. Wichtig ist nur, dass es *2* Variablen *gleichen* Typs sind. Die Funktion liefert dann einen Wert zurück, der ebenfalls dem Datentyp der Parameter entspricht.

```
template <typename T>
T maxi(T obj1, T obj2){
    if(obj1 > obj2)
        return obj1;
    else
        return obj2;
}
```

Das Template wird mit dem gleichnamigen aber kleingeschriebenen Schlüsselwort `template` eingeleitet. Danach folgt eine spitze Klammer auf und ein weiteres Schlüsselwort namens `typename`, früher hat man an dieser Stelle auch das gleichbedeutende Schlüsselwort `class` benutzt. Das funktioniert heute immer noch, ist aber nicht mehr üblich. Erstens Symbolisiert `typename` besser was folgt, nämlich ein Platzhalter, der für einen Datentyp steht und zweitens ist es einfach übersichtlicher. Dies gilt insbesondere bei Klassentemplates, die aber Thema des nächsten Kapitels sind.

Als nächstes folgt wie schon gesagt der Platzhalter. Er kann die gleichen Namen haben wie eine Variable sie haben könnte. Es ist aber allgemein üblich den Platzhalter als `T` (für Typ) zu bezeichnen. Das soll natürlich nur ein Vorschlag sein, viele Programmierer haben da auch ihren eigenen Stil. Wichtig ist nur, dass Sie sich nicht zu viele Bezeichner suchen, es sei denn Sie haben ein ausgesprochenes Talent dafür aussagekräftige Namen zu finden. Am Ende dieser Einführung wird die spitze Klammer geschlossen. Dann wird ganz normal die Funktion geschrieben, mit der eine Ausnahme, dass anstatt eines genauen Datentypes einfach der Platzhalter angegeben wird.

Sie können diese Funktion dann z.B. mit `int`-Werten aufrufen:

```
int a = 3, b = 5;
int m = maxi(a, b);                    // m == 5
m = maxi(3, 7);                        // m == 7
```

oder mit float-Werten:

```
float a = 3.3, b = 3.4;
float m = maxi(a, b);                  // m == 3.4
m = maxi(a, 6.33);                     // m == 6.33
```

Bei jedem ersten Aufruf einer solchen Funktion erstellt der Compiler aus dem Template eine echte Funktion, wobei er den Platzhalter durch den tatsächlichen Datentyp ersetzt. Welcher Datentyp das ist, entscheidet der Compiler anhand der übergebenen Argumente. Bei diesem Aufruf findet keine Typumwandlung statt, es ist also nicht möglich einen `int`-Wert und einen `float`-Wert an die Templatefunktion zu übergeben. Das nächste Beispiel wird das verdeutlichen:

```
int   a = 3;
```

```
float b = 5.4;
float m;                        // Ergebnis
m = maxi(a, b);                 // funktioniert nicht
m = maxi(4.5, 3);               // funktioniert nicht
m = maxi(4.5, 3.0);             // funktioniert
m = maxi(a, 4.8);               // funktioniert nicht
m = maxi(7, b);                 // funktioniert nicht
m = maxi(a, 5);                 // funktioniert
m = maxi(7.0, b);               // funktioniert
```

Möchten Sie eine andere Variante, als jene die anhand der Argumente aufgerufen wird benutzen, müssen Sie die gewünschte Variante explizit angeben:

```
float a = 3.0, b = 5.4;
float m = maxi<int>(a, b);            // m == 5.0
int   n = maxi<int>(a, b);            // m == 5
```

Diese Templatefunktion kann nun mit allen Datentypen aufgerufen werden, die sich nach der Ersetzung des Platzhalters durch den konkreten Datentyp auch übersetzen lassen. In unserem Fall wäre die einzige Voraussetzung, dass auf den Datentyp der Operator > angewandt werden kann.

Das ist beispielsweise auch bei C++-Strings (also Objekte des Klasse `std::string`, Headerdatei: `string`) der Fall. Daher ist folgendes problemlos möglich:

```
std::string s1("alpha");
std::string s2("omega");
std::string smax = maxi(s1,s2);       //smax == omega
```

46.1 Spezialisierung

Im Gegensatz zu normalen Funktionen können Templatefunktionen nicht einfach überladen werden, da das Template ja eine Schablone für Funktionen ist und somit bereits für *jeden* Datentyp eine Überladung bereitstellt. Manchmal ist es aber so,

dass das allgemein gehaltene Template für bestimmte Datentypen keine sinnvolle Funktion erzeugt. In unserem Beispiel würde dies für C-Strings zutreffen:

```
const char* maxi(const char* str1, const char* str2){
    if(str1 > str2)
        return str1;
    else
        return str2;
}
```

Beim Aufruf von `maxi("Ich bin ein String!", "Ich auch!")` würde die oben stehende Funktion aus dem Template erzeugt. Das Ergebnis wäre aber nicht wie gewünscht der dem ASCII-Code nach kleinere String, sondern einfach der String, dessen Adresse im Arbeitsspeicher kleiner ist. Um C-Strings zu vergleichen gibt es eine Funktion in der Headerdatei `cstring` (oder in der veralteten Fassung `string.h`).

Um nun dem Compiler beizubringen wie er `maxi()` für C-Strings zu implementieren hat, muss diese Variante spezialisiert werden. Das heißt, es wird für einen bestimmten Datentyp (in diesem Fall `const char*`) eine von der Vorlage (also dem Template) abweichende Version definiert. Für das nächste Beispiel muss die Headerdatei `cstring` inkludiert werden.

```
template <>
const char* maxi(const char* str1, const char* str2){
    if(strcmp(str1, str2) > 0) //strcmp Vergleicht 2 C-Strings
        return str1;
    else
        return str2;
}
```

Eine Spezialisierung leitet man immer mit dem Schlüsselwort `template` ein, gefolgt von einer öffnenden und einer schließenden spitzen Klammer. Der Name der Funktion ist identisch mit dem des Templates (`maxi`) und alle `T` müssen durch den konkreten Typ (`T` wird zu `const char*`) ersetzt werden. Der Rumpf der Funktion kann dann völlig beliebig gestaltet werden. Natürlich ist es Sinn und Zweck der Sache ihn so zu schreiben das die Spezialisierung das gewünschte sinnvolle Ergebnis liefert.

Übrigens wäre es auch möglich den als Beispiel dienenden Aufruf von oben als `maxi<std::string>("Ich bin ein String!", "Ich auch!")` zu schreiben. Dann würden die beiden C-Strings vor dem Aufruf in C++-Strings umgewandelt und die können ja Problemlos mit > verglichen werden. Diese Methode hat aber 3 entscheidende Nachteile gegenüber einer Spezialisierung für C-Strings:

- Die Headerdatei `string` muss jedes mal inkludiert werden
- Der Aufruf von `maxi("Ich bin ein String!", "Ich auch!")` ist weiterhin problemlos möglich und liefert auch weiterhin ein dem Zufall überlassenes Ergebnis
- Diese Art der Lösung ist viel langsamer als die Variante mit der Spezialisierung

46.2 Zwischen Spezialisierung und Überladung

Die folgende *Überladung* der Templatefunktion `maxi()` ist ebenfalls problemlos möglich und scheint auf den ersten Blick prima zu funktionieren:

```
//template <> - Ohne das ist es keine Spezialisierung sondern eine Überladung
const char* maxi(const char* str1, const char* str2){
    if(strcmp(str1, str2) > 0)
        return str1;
    else
        return str2;
}
```

Hinweis
Um es gleich vorwegzunehmen, das hier beschriebene Problem lässt sich leicht herbei führen, ist aber nur sehr schwer zu finden, daher achten Sie darauf es zu vermeiden. Vergessen Sie auf keinen Fall `template <>` voranzustellen, wenn Sie ein Template spezialisieren.

Das Ausführen der nächsten beiden Codezeilen führt zum Aufruf verschiedener Funktionen. Die Funktionen haben identische Parametertypen und Namen, sowie die gleiche Parameteranzahl, aber eben *nicht* den gleichen Code.

```
//Überladen Version - Ergebnis korrekt
cout << maxi("Ich bin ein String!", "Ich auch!");
//Vom Template erzeugte Version - Ergebnis zufällig (Speicheradressen)
cout << maxi<const char*>("Ich bin ein String!", "Ich auch!");
```

So etwas sollten Sie um jeden Preis vermeiden! Selbst ein erfahrener Programmierer dürfte über einen Fehler dieser Art erst einmal sehr erstaunt sein. Sie können die beiden Codezeilen ja mal jemandem zeigen der sich gut auskennt und ihn Fragen was der Grund dafür sein könnte, dass die Varianten 2 verschiedene Ergebnisse liefern. Ohne die Implementierung von `maxi()` zu sehen, wird er höchst wahrscheinlich nicht darauf kommen. Ein solcher Fehler ist nicht einfach nur schwer zu finden, es ist bereits eine Katastrophe ihn überhaupt erst einmal zu machen. Merken Sie sich also unbedingt, dass man `template <>` nur vor eine Funktion schreibt, wenn man für einen bestimmten Datentyp die Funktionsweise ändern will, weil die Standardvorlage kein sinnvolles Ergebnis liefert.

46.3 Überladen von Template-Funktionen

Ein Template ist (in der Regel) eine Vorlage für Funktionen, deren Parameter sich lediglich im Typ unterscheiden. Das Überladen von Funktionen ist aber auch durch eine andere Parameterzahl oder eine andere Anordnung der Parametertypen möglich. Unter dieser Überschrift wird die `maxi()`-Template um einen Optionalen dritten Parameter erweitert. Was bisher vorgestellt wurde wird in diesem Beispielprogramm zusammengefasst:

```
1. include <iostream> // Ein und Ausgabe
2. include <string> // C++-Strings
3. include <cstring> // Vergleiche von C-Strings


using namespace std;
// Unsere Standardvorlage für maxi()
template <typename T>
T maxi(T obj1, T obj2){
    if(obj1 > obj2)
        return obj1;
    else
        return obj2;
```

```
}
// Die Spezialisierungen für C-Strings
template <>                                          // Spezialisierung
const char* maxi(const char* str1, const char* str2){ // statt T, const char*
    if(strcmp(str1, str2) > 0)
        return str1;
    else
        return str2;
}
int main(){
    int   a = 3;                        // Ganzzahlige Variable
    float b = 5.4;                      // Gleitkommazahl-Variable
    int   m;                            // Ganzzahl-Ergebniss
    float n;                            // Kommazahl-Ergebniss
    // Je nachdem wo die Typumwandlungen stattfinden,
    // sind die Ergebnisse unterschiedlich
    m = maxi<int>(a, b);                // m == 5.0
    n = maxi<int>(a, b);                // n == 5
    m = maxi<float>(a, b);              // m == 5
    n = maxi<float>(a, b);              // n == 5.4
    // Aufgerufen wird unabhänngig vom Ergebnistyp m:
    m = maxi(3, 6);                     // maxi<int>()
    m = maxi(3.0, 6.0);                 // maxi<float>()
    m = maxi<int>(3.0, 6);              // maxi<int>()
    m = maxi<float>(3, 6.0);            // maxi<float>()
    // Aufruf von maxi<std::String>()
    string s1("alpha");
    string s2("omega");
    string smax = maxi(s1,s2);          // smax == omega
    // Aufruf von maxi<const char*>()
    smax = maxi("alpha","omega");       // Spezialisierung wird aufgerufen
    return 0;                           // Programm erfolgreich durchlaufen
}
```

Eine Ausgabe enthält das Programm noch nicht, aber das sollten Sie ja problemlos schaffen. Die Datei `iostream` ist auch schon inkludiert.

Nun aber zur Überladung der Templatefunktion. Um von `maxi()` zu verlangen, den größten von 3 Werten zurückzugeben, muss der Parameterliste ein weiterer Platzhalter (also noch ein `T obj3`) hinzugefügt werden. Gleiches gilt für unsere C-String Spezialisierung. Da wir ja aber auch die Möglichkeit haben wollen, nur den

größeren von 2 Werten zu bestimmen, müssen wir die bisherige Variante stehen lassen und die mit 3 Argumenten hinzufügen. Das ganze sieht dann so aus:

```
    1. include <iostream> // Ein und Ausgabe
    2. include <string> // C++-Strings
    3. include <cstring> // Vergleiche von C-Strings


using namespace std;
// Unsere Standardvorlage für maxi() mit 2 Argumenten
template <typename T>
T maxi(T obj1, T obj2){
  if (obj1 > obj2)
    return obj1;
  else
    return obj2;
}
// Die Spezialisierungen für C-Strings mit 2 Argumenten
template <> // Spezialisierung
const char* maxi(const char* str1, const char* str2){
    if(strcmp(str1, str2) > 0)
        return str1;
    else
        return str2;
}
// Unsere Standardvorlage für maxi() mit 3 Argumenten
template <typename T>
T maxi(T obj1, T obj2, T obj3){
    if(obj1 > obj2)
        if(obj1 > obj3)
            return obj1;
        else
            return obj3;
    else if(obj2 > obj3)
        return obj2;
    else
        return obj3;
}
// Die Spezialisierungen für C-Strings mit 3 Argumenten
template <> // Spezialisierung
const char* maxi(const char* str1, const char* str2, const char* str3){
    if(strcmp(str1, str2) > 0)
```

```
        if(strcmp(str1, str3) > 0)
            return str1;
        else
            return str3;
    else if(strcmp(str2, str3) > 0)
        return str2;
    else
        return str3;
}
int main(){
    cout << "Beispiele für Ganzzahlen:\n";
    cout << "      (2, 7, 4) -> " << maxi(2, 7, 4) << endl;
    cout << "         (2, 7) -> " << maxi(2, 7)    << endl;
    cout << "         (7, 4) -> " << maxi(7, 4)    << endl;
    cout << "\nBeispiele für Kommazahlen:\n";
    cout << "(5.7, 3.3, 8.1) -> " << maxi(5.7, 3.3, 8.1) << endl;
    cout << "     (7.7, 7.6) -> " << maxi(7.7, 7.6)      << endl;
    cout << "     (1.9, 0.4) -> " << maxi(1.9, 0.4)      << endl;
    cout << "\nBeispiele für C-Strings:\n";
    cout << "(ghi, abc, def) -> " << maxi("ghi", "abc", "def") << endl;
    cout << "     (ABC, abc) -> " << maxi("ABC", "abc")        << endl;
    cout << "     (def, abc) -> " << maxi("def", "abc")        << endl;
    return 0;                            // Programm erfolgreich durchlaufen
}
```

Ausgabe

```
Beispiele für Ganzzahlen:
      (2, 7, 4) -> 7
         (2, 7) -> 7
         (7, 4) -> 7
Beispiele für Kommazahlen:
(5.7, 3.3, 8.1) -> 8.1
     (7.7, 7.6) -> 7.7
     (1.9, 0.4) -> 1.9
Beispiele für C-Strings:
(ghi, abc, def) -> ghi
     (ABC, abc) -> abc
     (def, abc) -> def
```

So Sie Verstanden haben, wie aus einem Template ein konkrete Funktion erzeugt wird, fragen Sie sich vielleicht, ob das wirklich so einfach ist. Glücklicherweise

kann diese Frage bejaht werden. Man stellt sich einfach das Funktionstemplate als mehrere separate Funktionen vor, die durch unterschiedliche Parametertypen überladen sind und wendet dann die üblichen Regeln zur Überladung von Funktionen an. Wenn man, wie in diesem Fall getan, ein weiteres Funktionstemplate anlegt, werden dadurch natürlich auch gleich wieder eine ganze Reihe einzelner Funktionen hinzugefügt. Im Endeffekt überlädt man also nicht das Template, sondern immer noch die aus ihm erzeugten Funktionen. Was Spezialisierungen an geht, die gehören natürlich immer zu dem Template, das eine zu seiner semantische Parameterliste besitzt.

46.4 Templates mit mehreren Parametern

Natürlich kann ein Template auch mehr als nur einen Templateparameter übernehmen. Nehmen Sie an, Sie benötigen eine kleine Funktion welche die Summe zweier Argumente bildet. Über den Typ der Argumente wissen Sie nichts, aber der Rückgabetyp ist Ihnen bekannt. Eine solche Funktion könnte folgendermaßen Implementiert werden:

```
    1. include <iostream> // Ein und Ausgabe


using namespace std;
// Unsere Vorlage für die Summenberechnung
template <typename R, typename Arg1, typename Arg2>
R summe(Arg1 obj1, Arg2 obj2){
    return obj1 + obj2;
}
int main(){
    float a = 5.4;                      // Ganzzahlige Variable
    int   b = -3;                       // Gleitkommazahl-Variable
    cout << summe<int>(a, b)             << endl; // int           summe(float obj1, int obj2);
    cout << summe<float>(a, b)           << endl; // float         summe(float obj1, int obj2);
    cout << summe<double>(a, b)          << endl; // double        summe(float obj1, int obj2);
    cout << summe<unsigned long>(a, b)   << endl; // unsigned long summe(float obj1, int obj2);
    cout << summe<float, int, int>(a, b) << endl; // float         summe(int   obj1, int obj2);
    cout << summe<float, long>(a, b)     << endl; // float         summe(long  obj1, int obj2);
    return 0;                            // Programm erfolgreich durchlaufen
}
```

Ausgabe

```
2
```

```
2.4
2.4
2
2
2
```

Beim fünften Aufruf sind alle Argumente explizit angegeben. Wie Sie unschwer erkennen werden, kann der Compiler bei dieser Funktion den Rückgabetyp nicht anhand der übergeben Funktionsparameter ermitteln, daher müssen Sie ihn immer explizit angeben, wie die ersten 4 Aufrufe es Zeigen.

Die Templateargumente werden beim Funktionsaufruf in der gleichen Reihenfolge angegeben, wie sie bei der Deklaraton des Templates in der Templatearumenliste stehen. Wenn dem Compiler weniger Argumente übergeben werden als bei der Deklaration angegeben, versucht er die übrigen anhand der Funktionsargumente zu bestimmen.

Das bedeutet für Sie, das Argumente die immer explizit angegeben werden müssen, auch bei der Deklaration an vorderster Stelle der Templateargumentliste stehen müssen. Das folgende Beispiel verdeutlicht die folgen bei Nichteinhaltung dieser Regel:

```
    1. include <iostream> // Ein und Ausgabe


using namespace std;
// Unsere Vorlage für die Summenberechnung
template <typename Arg1, typename Arg2, typename R> // Rückgabetyp als letztes angegeben
R summe(Arg1 obj1, Arg2 obj2){
    return obj1 + obj2;
}
int main(){
    float a = 5.4;                      // Ganzzahlige Variable
    int   b = -3;                       // Gleitkommazahl-Variable
    cout << summe<int>(a, b)            << endl; // ??? summe(int           obj1, int  obj2);
    cout << summe<float>(a, b)          << endl; // ??? summe(float         obj1, int  obj2);
    cout << summe<double>(a, b)         << endl; // ??? summe(double        obj1, int  obj2);
    cout << summe<unsigned long>(a, b)  << endl; // ??? summe(unsigned long obj1, int  obj2);
    cout << summe<float, int, int>(a, b) << endl; // int summe(float         obj1, int  obj2);
    cout << summe<float, long>(a, b)    << endl; // ??? summe(float         obj1, long obj2);
    return 0;                           // Programm erfolgreich durchlaufen
```

```
}
```

Nur der Aufruf (Nr. 5) bei dem auch das letzte Tempalteargument (also der Rückgabetyp) explizit angegeben wurde ließe sich übersetzen, alle anderen bemängelt der Compiler. Sie müssten also alle Templateargumente angeben um den Rückgabetyp an den Compiler mitzuteilen. Das schmeißt diese wundervolle Gabe des Compilers, selbständig anhand der Funktionsparameter die richtige Templatefunktion zu erkennen, völlig über den Haufen. Hinzu kommt in diesem Beispiel noch, dass die Reihenfolge bei einer expliziten Templateargumentangabe alles andere als intuitiv ist.

46.5 Nichttypargumente

Neben Typargumenten können Templats auch ganzzahlige Konstanten als Argument übernehmen. Die folgende kleine Templatefunktion soll alle Elemente eines Arrays beliebiger Größe mit einem Startwert initialisieren.

```
   1. include <cstddef> // Für den Typ size_t


template <std::size_t N, typename T>
void array_init(T (&array)[N], T const &startwert){
    for(std::size_t i=0; i!=N; ++i)
        array[i]=startwert;
}
```

Die Funktion übernimmt eine Referenz auf ein Array vom Typ `T` und mit `N` Elementen. Jedem Element wird der Wert zugewiesen, welcher der Funktion als zweites Übergeben wird. Der Typ `std::size_t` wird übrigens von der C++-Standardbibliothek zur Verfügung gestellt. Er ist in der Regel als `typedef` auf `long` deklariert.

Ihr Compiler sollte in der Lage sein, sowohl den Typ `T`, als auch die Größe des Arrays `N` selbst zu ermitteln. In einigen Fällen kann es jedoch vorkommen das der Compiler nicht in der Lage ist die Größe `N` zu ermitteln, da sie als erstes Templateargument angegeben wurde, reicht es aus nur sie explizit anzugeben und den Compiler in diesen Fällen zumindest den Typ `T` selbst ermitteln zu lassen.

Kapitel 47

Container

47.1 Kontainerklassen

In diesem Kapitel soll das Konzept generischer Kontainer vorgestellt werden. Die C++ Standardbibliothek stellt folgende Typen zur Verfügung:

Sequentielle Kontainerklassen

- vector
- deque
- list

Assoziative Kontainerklassen

- hash_map
- hash_multimap
- hash_multiset
- hash_set
- map
- multimap
- multiset
- set

Kontaineradapterklassen (keine C++ Standardbibliothek-Iteratoren)

- priority_queue
- queue
- stack

Container sind Behälter für Objekte. Dabei wird immer eine Kopie des Objektes in dem Container gespeichert. Das hat den Vorteil, das sich die Speicherverwaltung vereinfacht, denn das Objekt verliert seine Gültigkeit, wenn das Containerobjekt aufgelöst wird. Der Nachteil liegt auf der Hand - durch das Erstellen einer Kopie geht Zeit und Speicherplatz verloren. Es gibt aber auch Wege Zeiger in Containern zu speichern, was aber an späterer Stelle erläutert werden soll. Ein weiterer Vorteil bei der Verwendung von Containern ist ihre einheitliche Schnittstelle zu Elementfunktionen. Das vereinfacht den Einsatz von bestimmten Algorithmen, wie sie ebenfalls in der STL zu finden sind.

47.1.1 Kontainereigenschaften

Ein *vector* ist im Wesentlichen ein dynamisches Feld, das je nach Bedarf seine Größe dynamisch verändern kann. Allerdings sind dabei einige Dinge zu beachten. Auf ein beliebiges Objekt innerhalb des Feldes kann sehr effizient unter direkter Adressierung zugegriffen werden - man spricht daher auch von konstantem Aufwand, da sich die Zugriffszeit nicht ändert, wenn das Feldobjekt wächst. Ebenso ist das Anhängen und Entfernen von Objekten am Ende effizient, nicht so hingegen am Anfang oder in der Mitte (linearer Aufwand). Der Container *deque* beherrscht ebenso wie *vector* das schnelle Einfügen von Objekten am Ende und dazu noch am Anfang des Feldes. Hingegen ist auch hier das Einfügen von Objekten in der Mitte sehr aufwendig. Auch dieser Containertyp unterstützt *Random-Access*-Operatoren, d.h. der Zugriff auf ein beliebiges Element ist sehr effizient. Der dritte sequenzielle Container ist der *list*-Container. Dieser Typ unterstützt nur sogenannte *Bidirectional*-Iteratoren. Dadurch ist ein direkter Indexzugriff wie bei den anderen Containern nicht möglich. Der Vorteil dieses Containers ist allerdings des effiziente Einfügen und Entfernen von Objekten an beliebigen Positionen des Feldes. Durch diese deutlichen Unterschiede sollte sich der Programmierer schon bei der Planung des Programms darüber im Klaren sein, welcher Containertyp am besten zu seinen Anforderungen passt.

47.1.2 Grundlegende Mitgliedsfunktionen der sequentiellen Kontainerklassen

Sequenzielle Container besitzen Funktionen wie `front()` oder auch `back()`. Damit kann auf das erste bzw. letzte Element des Containers zugegriffen werden. Darüber hinaus gibt es Funktionen, die das Anhängen und Entfernen von Elementen am Ende eines Feldes erlauben (`push_back()`, `pop_back()`). Das Einfügen/Entfernen von Objekten am Anfang wird nur von den Typen *deque* und *list* beherrscht (mittels: `push_front()` und `pop_front()`). Das direkte Indizieren beherrschen *vector* und *deque* (mittels `at()` bzw. mit `operator[]`). Die folgende Tabelle soll dies vergleichend veranschaulichen.

	vector	deque	list
front()	x	x	x
back()	x	x	x
push_back()	x	x	x
pop_back()	x	x	x
push_front()	/	x	x
pop_front()	/	x	x
at()	x	x	/
operator[]	x	x	/

Darüber hinaus sind weitere Funktionen definiert, deren konkrete Implementierung vom jeweiligen Containertyp abhängig sind. Als Beispiel sei hier der *vector*-Container gewählt.

Funktionsname	**Beschreibung**
assign	Zuweisen von Elementen zum vector
begin	gibt einen Iterator auf das erste Element zurück
capacity	gibt Anzahl an Elementen zurück die vom vector aufgenommen werden können
clear	löscht alle Elemente
empty	gibt *wahr* zurück wenn Container leer ist
end	gibt einen Iterator auf das letzte Element zurück
erase	löscht das Element oder eine Reihe von Elementen

insert	fügt Elemente in den Container ein
max_size	gibt die maximal mögliche Zahl von Elementen des Containers an
rbegin	gibt einen *reverse* Iterator auf den Anfang zurück
rend	gibt einen *reverse* Iterator auf das Ende zurück
reserve	setzt eine minimale Kapazität des Containers
resize	ändert die Größe des Containers
size	gibt aktuelle Größe des Feldes zurück
swap	tauscht den Inhalt des Containers mit einem anderen

47.1.3 Adaptive Kontainer

Dies sind spezielle Container, die auch Containeradapter genannt werden. Insbesondere lassen sich auf diese Container keine Iteratoren definieren. Der Zugriff erfolgt ausschließlich über die Elementfunktionen dieser Typen. In der STL kann man die Typen

- stack
- queue
- priority_queue

finden.

Kapitel 48

Interne Zahlendarstellung

Wie bereits in der Einführung der Variablen erwähnt, sind Variablen nichts weiter als eine bequeme Schreibweise für eine bestimmte Speicherzelle bzw. eine Block von Speicherzellen. Die kleinste Speicherzelle, welche direkt vom Prozessor "adressiert" werden kann, ist ein Byte. Ein Byte besteht (heutzutage) aus 8 Bit. Meist stellt man ein Byte grafisch so dar:

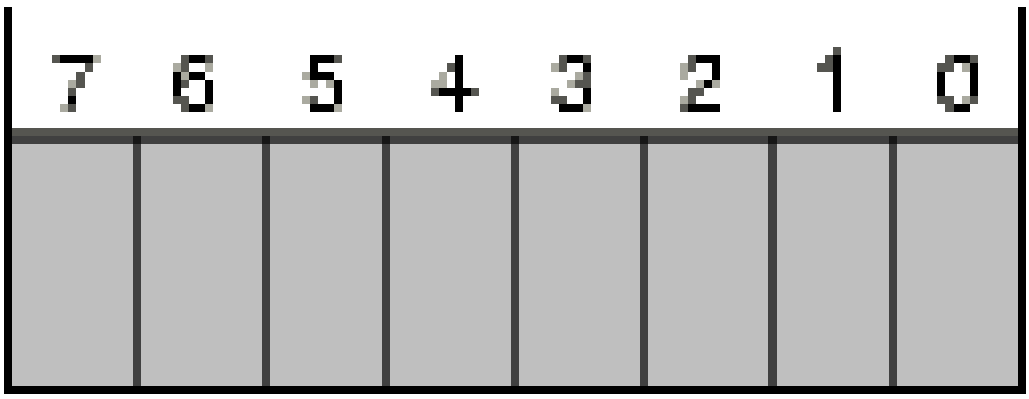

Abbildung 3

Das unterste Bit, Bit 0, wird als "niederwertigstes Bit" bezeichnet (englisch: "least significant bit: LSB"), das oberste Bit, Bit 7, als "höchstwertiges Bit" (englisch: "most significant bit: MSB").

Jedes dieser acht Bits kann den Wert 0 oder 1 annehmen. Damit kann ein Byte 2^8=256 verschiedene Werte annehmen: Wie diese 256 verschiedenen Werte "interpretiert" werden, ist abhängig vom Datentyp. Der C++ Datentyp, der genau

ein Byte repräsentiert, ist char, also zu Deutsch: "Zeichen". Jeder Buchstabe, jede Ziffer, jedes der so genannten "Sonderzeichen" kann in einem solchen char gespeichert werden. (Zumindestens sofern man in Westeuropa bleibt. Für die vielen verschiedenen Zeichen z.B. der asiatischen Sprachen genügt ein char nicht mehr, doch dafür gibt es den Datentyp wchar_t, der größer als ein Byte ist und somit mehr Zeichen darstellen kann.)

Das Zeichen 'A' wird z.B. durch die Bitkombination 01000001 dargestellt, das Zeichen '&' durch 00100110. Die Zuordnung von Bitwerten zu Zeichen ist im (ASCII-Code) festgeschrieben, den heute eigentlich alle Computer verwenden. Im Anhang D ist der ASCII-Code aufgelistet.

Die Bitkombinationen eines Bytes kann man natürlich auch als Zahlen auffassen. Die Bitkombinationen 00000000 bis 01111111 entsprechen dann den Zahlen 0 bis 127. Leider ist nun die Frage, welche Zahlen den Bitfolgen 10000000 bis 11111111 entspricht, nicht mehr so eindeutig. Naheliegend wären ja die Zahlen 128 bis 255. Es könnten aber auch die Zahlen -128 bis -1 sein. Um die zugrunde liegenden Zusammenhänge zu erläutern muss etwas weiter ausgeholt werden

Wie auch bei den Ganzzahltypen (und char gehört schon ein wenig mit dazu), unterscheidet man zwischen "vorzeichenbehafteten" (englisch: "signed") und "vorzeichenlosen" (englisch: "unsigned") Zahlen. Diese Unterscheidung existiert jedoch nur im "Kopf des Programmierers". Der Prozessor weiß nicht, ob der Wert einer Speicherzelle vorzeichenbehaftet ist oder nicht. Zum Addieren und Subtrahieren muss er das auch nicht wissen. Die gewählte "Zweierkomplement-Darstellung" gestattet es, mit vorzeichenbehafteten Zahlen genauso zu rechnen wie mit vorzeichenlosen. Nur beim Multiplizieren und Dividieren muss man dem Prozessor mitteilen, ob er eine normale (vorzeichenlose) oder eine vorzeichenbehaftete Operation ausführen soll.

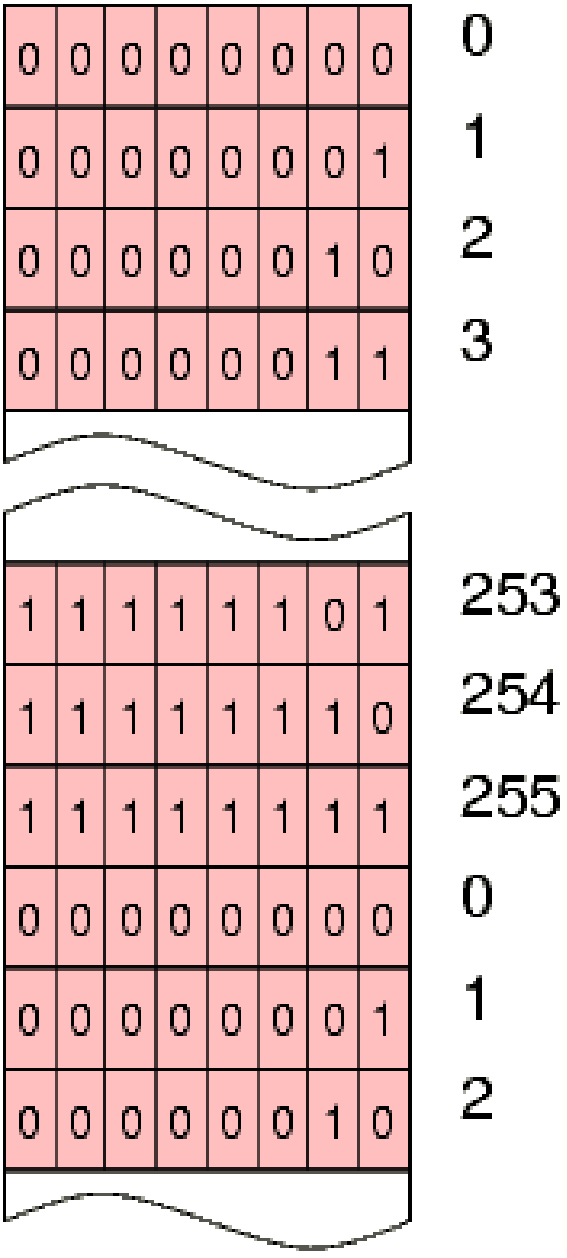
0 0 0 0 0 0 0 0
0
0 0 0 0 0 0 0 1
1
0 0 0 0 0 0 1 0
2
0 0 0 0 0 0 1 1
3
1 1 1 1 1 1 0 1
253
1 1 1 1 1 1 1 0
254
1 1 1 1 1 1 1 1
255
0 0 0 0 0 0 0 0
0
0 0 0 0 0 0 0 1
1
0 0 0 0 0 0 1 0
2

Abbildung 4

Dezimal	**Bit-Stellen**								
		7	6	5	4	3	2	1	0
...	...								
252		1	1	1	1	1	1	0	0
253		1	1	1	1	1	1	0	1
254		1	1	1	1	1	1	1	0
255		1	1	1	1	1	1	1	1
256	1	0	0	0	0	0	0	0	0
257	1	0	0	0	0	0	0	0	1
258	1	0	0	0	0	0	0	1	0
259	1	0	0	0	0	0	0	1	1
...	...								

Zur besseren Veranschaulichung die beispielhafte Binärdarstellung der Zahlen um 255 herum:

Sie sehen, dass die Zahlen ab 256 nicht mehr in die 8 Bits eines Bytes passen, das Byte "läuft über". Wenn man nun z.B. zu 255 eine 1 addiert, und das Ergebnis in einem Byte speichern will, wird das oberste Bit abgeschnitten und übrigt bleibt eine 0, die dann gespeichert wird.

Da bei dem Übergang von 255 zu 0 die Zahlen wieder von vorne anfangen, kann man sich den Zahlenbereich auch als Kreis vorstellen. Entegenen dem Uhrzeigersinn (in Pfeilrichtung) "wachsen" die Zahlen, in die andere Richtung fallen sie. An irgend einer Stelle erfolgt dann ein "Sprung". Wenn diese Stelle bei einer Berechnung überschritten wird, nennt man dies einen "Überlauf". Bei "unsigned" Zahlen unterhalb der 0, bei den "signed" Zahlen genau gegenüber:

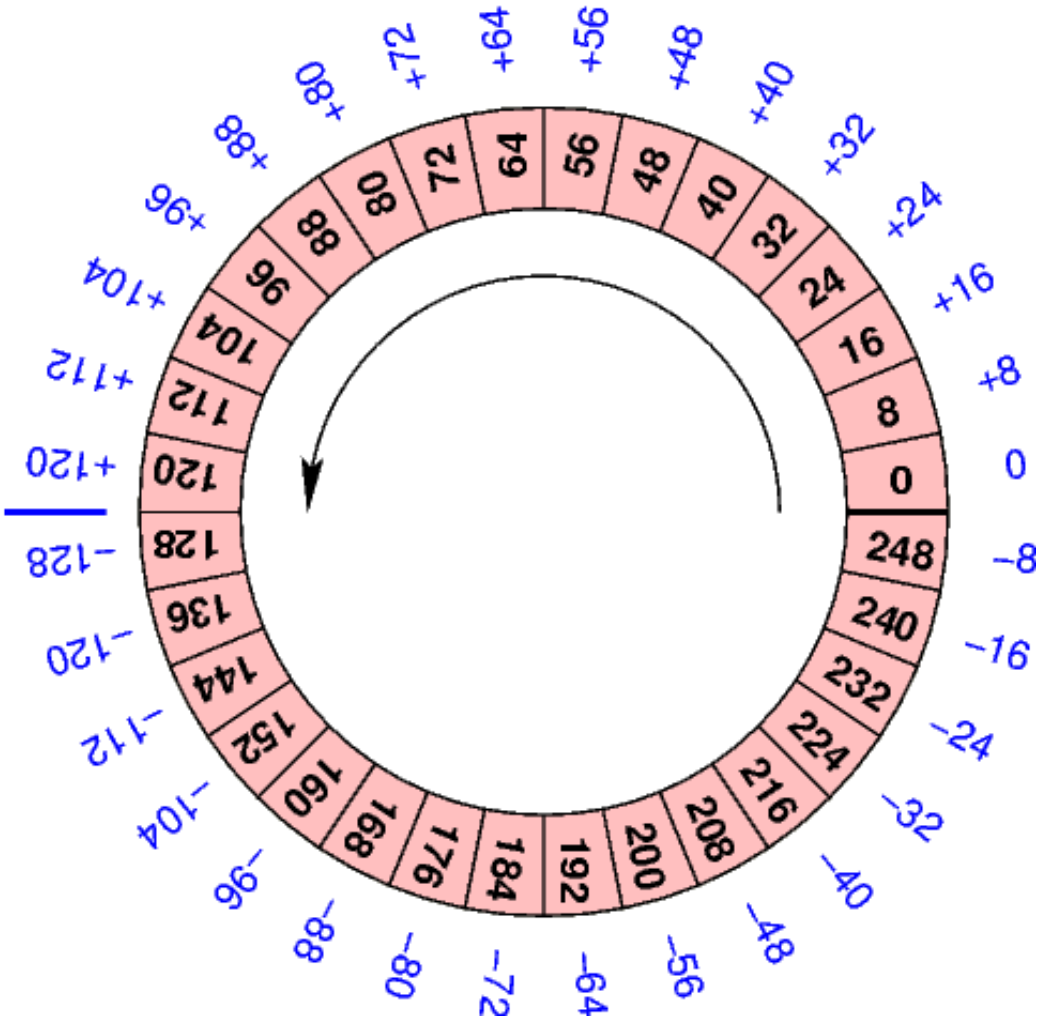

Abbildung 5

Diese Darstellung negativer Zahlen hat einige Vorteile:

1. Der Prozessor muss beim Addieren und Subtrahieren nicht darauf achten, ob eine Zahl vom Programmierer als vorzeichenbehaftet oder vorzeichenlos interpretiert wird. Beim Addieren zählt er entsprechend viele Zahlen "links herum", beim Subtrahieren entsprechend andersrum. Falls dabei ein Überaluf auftritt, ist das dem Prozessor "egal".
2. Die "Sprungstelle", also der Überlauf, ist möglichst weit weg von der Null und den kleinen Zahlen, die man am meisten benutzt, so dass man beim Rechnen (hoffentlich) nicht in den Bereich dieses Überlaufes kommt.

Diese Darstellung negativer Zahlen wird auch bei den anderen Ganzzahltypen angewendet, nur ist der Kreis dort wesentlich größer, und enthält mehr darstellbare Zahlen.

Während nun bei den Ganzzahltypen int, short und long festgelegt ist, dass sie standardmäßig immer "signed" sind (so kann man in C++ auch signed int statt int usw. schreiben, es ist der gleiche Datentyp), konnte man sich bei char nicht einigen, ob mit oder ohne Vorzeichen praktischer ist. Folglich kann jeder Compilerhersteller dies nach eigenem Gusto entscheiden. Ob auf einem System der char-Datentyp nun vorzeichenbehaftet ist oder nicht, ist jedoch in der Praxis nicht wichtig, sofern man nicht mit ihnen Rechnen will. (Zum Rechnen werden char-Variablen und -Literale stets erst in ein int umgewandelt. Erst dann wird z.B. aus einem Byte mit den Bits 11111111 der Wert 255 oder -1, je nach dem.)

Sie können jedoch explizit signed char oder unsigned char schreiben, wenn Sie sicher gehen wollen, dass Sie ein vorzeichenbehaftetes bzw. vorzeichenloses char bekommen. (Kurioserweise ist ein char ein eigener Datentyp, auch wenn er stets entweder einem signed char oder einem unsigned char gleicht. Es gibt also in C++ drei verschiedene char-Datentypen!)

Die übrigen Ganzzahltypen belegen mehrere Bytes, wie viele genau, kann man herausbekommen, in dem man mal ein kleines Programm schreibt, das folgendes ausgibt:

```
std::cout << "Größe eines short: " << sizeof(short) << std::end;
std::cout << "Größe eines int  : " << sizeof(int)   << std::end;
std::cout << "Größe eines long : " << sizeof(long)  << std::end;
```

sizeof(Typ_oder_Ausdruck) gibt die Größe eines Typs oder Ausdrucks in Byte zurück. (Genauer: sizeof() gibt an, wie viele Bytes der Datentyp im Vergleich zu einem ’char’ benötigt. sizeof(char) ist definitionsgemäß gleich 1, da keine kleinere Speichergröße direkt angesprochen werden kann.)

Die Größe der Datentypen ist jedoch nicht genau festgelegt und variiert von Compiler zu Compiler und von Prozessor zu Prozessor! Man darf sich also nicht darauf verlassen, dass ein int immer soundsoviel Bytes groß ist! Das einzige, das im C++ Standard festgelegt ist, ist folgende Größenrelation:

```
sizeof(char) := 1
sizeof(long) ≥ sizeof(int) ≥ sizeof(short) ≥ 1
sizeof( X ) = sizeof( signed X ) = sizeof( unsigned X )    (für X = int, short, long, char)
```

Weiß man jedoch die Größe eines Ganzzahltyps, weiß man auch, wie groß der Wertebereich ist, den er aufnehmen kann:

Größe in Byte	**Wertebereich**	
	`unsigned`	`signed`
1 (8bit)	0 … 255	-128 … +127
2 (16bit)	0 … 65.536	-32.768 … +32.767
4 (32bit)	0 … 4.294.967.296	-2.147.483.648 … +2.147.483.647
8 (64bit)	0 … 18.446.744.073.709.551.616	-9.223.372.036.854.775.808 … +9.223.372.036.854.775.807

Warum hat man die Größe der Datentypen nicht festgelegt?

Ganz einfach. Es existieren sehr verschiedene Prozessortypen. Es gibt so genannte 16-Bit-Prozessoren, das heißt, diese können 16-bittige Zahlen "in einem Rutsch" verarbeiten. Heutige PC-Prozessoren sind meist 32-Bit-Prozessoren, sie können 32-Bit-Zahlen besonders schnell verarbeiten. Inzwischen kommen auch mehr und mehr 64-Bit-Prozessoren auf, vielleicht werden sie sich ja in ein paar Jahren die 32-Bit-Prozessoren verdrängen.

Hätte man jetzt festgelegt, dass ein int immer 32-bit groß ist, wären C++ Programme auf 16-Bit-Prozessoren sehr langsam, da sie nur "häppchenweise" mit 32-bit-Werten rechnen können. Darum hat man festgelegt, dass ein int immer so groß sein soll, wie die so genannte "Wortbreite" des Prozessors. Das garantiert, dass Berechnungen mit int's immer am schnellsten ausgeführt werden können.

Mal als Überblick, welche größen die Ganzzahl-Datentypen auf verschiedenen Plattformen haben:

Plattform	**Größe des Datentyps (in Byte)**		
	`short`	`int`	`long`
DOS (16 bit "Real Mode")	2	2	4
Linux (auf 32bit-Plattformen wie x86, PowerPC u.a.)OS/2 (ab 3.0)MS-Windows (ab 9x/NT/2k)	2	4	4
Linux (auf 64bit-Plattformen wie Alpha, amd64 u.a.)	2	4	8

Man sieht, dass unter Linux auch auf einem 64bit-System ein int nur 32 Bit groß ist. Vermutlich hat man sich für 32-bittige int's entschieden, weil es zu viele C- und C++-Programme gibt, die darauf vertrauen, dass ein int immer 32 bit groß ist. :-(Ich würde mich aber nicht darauf verlassen, dass das auch in Zukunft so bleiben wird.

Wenn man low-level Code schreiben will/muss, und auf die genaue Größe seiner Ganzzahltypen angewiesen ist, gibt es spezielle typedef's in dem Standard-Header <stdint.h>, auf welche später genauer eingegangen wird.

Zusammenfasend läßt sich also aussaagen, dass ein int mehrere auf einander folgende Bytes im Speicher belegt. Wie bereits erläutert, werden die Bits in einem Byte von 0 bis 7 gezählt, das Bit 0 ist dabei die "Einer-Stelle", das Bit 7 hat die Wertigkeit 128. (bzw. -128 bei vorzeichenbehafteten Bytes) Bei einem 2-Byte-short int ergeben sich also 16 Bits, und das Ganze sieht dann wie folgt aus:

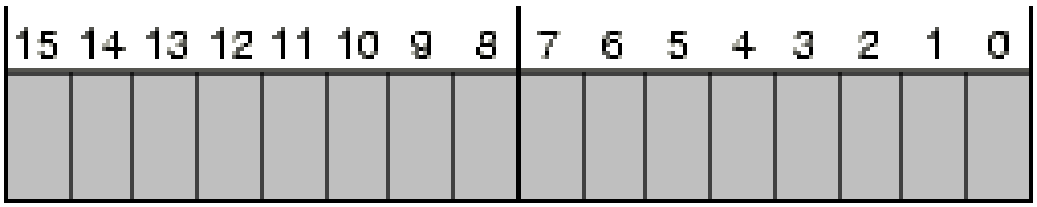

Abbildung 6

Das MSB ist in diesem Falle das Bit 15. Die Bits 0..7 bilden das so genannte "Low Byte", die Bits 8..15 das "High Byte".

So weit so gut. Das Problem ist jetzt die Byte-Reihenfolge, also wie dieses "Doppelbyte" abgespeichert wird. Die einen sagen: Low-Byte zuerst, also das "untere Byte" an die "kleinere Adresse", das High-Byte in die darauf folgende, dies nennt man "Little Endian". Andere finden es natürlicher, wenn das High-Byte "zuerst" gespeichert wird, also an der kleinere Adresse im Speicher liegt. Dies nennt sich "Big Endian"-Speicherung. Diese beiden Methoden der Speicherung von Mehrbyte-Werten existieren auch bei 32- und 64-Bit-Werten.

Da sich in dieser Frage die verschiedenen Prozessor-Hersteller nicht einig wurden, existieren bis heute beide Varianten. Der eine Prozessor macht es so herum, ein anderer anders herum:

Prozessor-Plattform	"Endianness"
Intel x86 und kompatible	Little Endian
Intel Itanium (64-Bit-CPU)	Little Endian
Motorola 68k	Big Endian
PowerPC	Vom Betriebssystem einstellbar
Sun SPARC	Big Endian
DEC Alpha	Vom Betriebssystem einstellbar

Wichtig wird diese so genannte "Byte order" immer dann, wenn man Daten in eine Datei speichern oder über ein Netzwerk auf andere Computer schicken will. Denn falls die Datei später von einem Computer mit anderer Byte Order gelesen wird, versteht er die gespeicherten Daten nicht mehr richtig. Man muss also bei der Speicherung oder Übermittlung von Daten stets vorher festlegen, welche Byte Order man benutzt. Im Internet hat man dafür den Begriff "network byte order" geprägt. Diese bedeutet: Big Endian.

48.1 Gleitkommazahlen

Gleitkommazahlen werden heutzutage meist nach dem so genannten IEEE 754 Standard abgespeichert. Nach diesem Standard ist ein 32bit- (einfache Genauigkeit) und ein 64bit- (doppelte Genauigkeit) Gleitkommadatentyp definiert. Der 32-bit Gleitkommadatentyp ist dabei wie folgt aufgebaut:

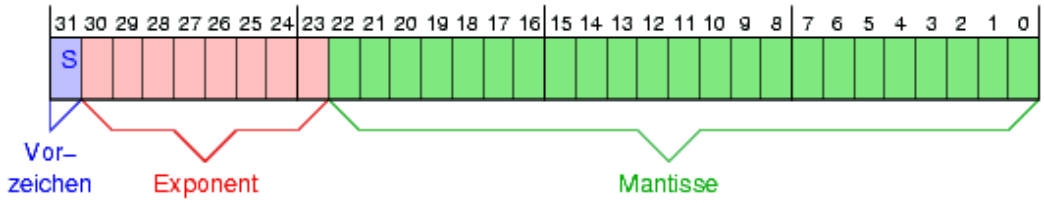

Abbildung 7

Das Vorzeichen-Bit gibt dabei an, ob die Zahl positiv (Vorzeichen=0) oder negativ (Vorzeichen=1) ist. In der Mantisse werden nur die Binärstellen nach dem Komma gespeichert. Vor dem Komma steht implizit stets eine 1. Der gespeicherte Wert für den Exponenten ist um 127 größer als der echte Exponent. Die gespeicherten Werte von 0..255 für den Exponenten entsprechen also einem realen Wertebereich von -127...+128. Dabei sind die Werte -127 und +128 "reserviert" für spezielle Gleitkommawerte wie "null" oder "unendlich" (und noch einige andere, so genannte NaN's und 'denormalisierte Zahlen', worauf ich hier aber nicht weiter eingehen möchte.) Der Wert einer 32-bit-Gleitkommazahl errechnet sich dabei wie folgt:

$$Wert = -1^V * 2^{(exp-127)} * 1.mantisse$$

Der Wert "null" wird dabei mit Exponent 0 und Mantisse 0 gespeichert. Dabei kann das Vorzeichen 0 oder 1 sein, es wird also zwischen +0.0 und -0.0 unterschieden. Damit kann man z.B. erkennen, ob das Ergebnis einer Berechnung ein "sehr kleiner positiver Wert" war, der auf 0 gerundet wurde, oder ein "sehr kleiner

negativer Wert". Ansonsten sind beide Werte identisch. Ein Vergleich (+0.0 == -0.0) ergibt also den Wert "true", also "wahr".

Der Wert "unendlich" wird mit Exponent 255 und Mantisse 0 gespeichert. "Unendlich" bedeutet dabei "größer als die größte darzustellende Zahl". Auch hier wird wieder zwischen "+unendlich" und "-unendlich" unterschieden. "Unendlich" kann dabei durchaus sinnvoll sein. So liefert die mathematische Funktion atan() (arcus tanges) für "unendlich" genau den Wert Pi/2.

Kapitel 49

Autoren

Edits	User
1	Axelf
1	Banco
12	Bernina
19	Bulldog98
3	Ce2
9	Chfreund
2	Cilania
1	Daniel B
17	Dirk Huenniger
17	E(nix)
2	Einfach Toll
6	Etlam
19	Extendable
43	Fencer
7	Gronau
2	Heuler06
1	Hjn
1	InselAX
1	Jan Luca
1	Jarling
1	K@rl
13	Klaus Eifert
2	Krje
2	Marlon93
10	Martin Fuchs

1 Matze
8 MetalHeart
1 MichaelFrey
5 MichaelFreyTool
2 Mr. Anderson
2 Muxxxa
2 Norro
2 O-Dog
1 Onegin
5 Philipp K.
425 Prog
6 Pumbaa80
1 Remohe
35 RokerHRO
13 Ronsn
9 Sliver
1 Steef389
11 Stefan Kögl
2 Szs
7 ThePacker
1 Therapiekind
1 Thomas J
1 Tschäfer
2 W4R
2 WikiBookPhil
2 Yeto

Kapitel 50

Bildnachweis

In der nachfolgenden Tabelle sind alle Bilder mit ihren Autoren und Lizenen aufgelistet.

Für die Namen der Lizenzen wurden folgende Abkürzungen verwendet:

- GFDL: Gnu Free Documentation License. Der Text dieser Lizenz ist in einem Kapitel diese Buches vollständig angegeben.
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/3.0/ nachgelesen werden.
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/2.5/ nachgelesen werden.
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/2.0/ nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by-sa/1.0/ nachgelesen werden.
- cc-by-2.0: Creative Commons Attribution 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by/2.0/ nachgelesen werden. Mit

dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.

- cc-by-2.5: Creative Commons Attribution 2.5 License. Der Text dieser Lizenz kann auf der Webseite http://creativecommons.org/licenses/by/2.5/deed.en nachgelesen werden.
- GPL: GNU General Public License Version 2. Der Text dieser Lizenz kann auf der Webseite http://www.gnu.org/licenses/gpl-2.0.txt nachgelesen werden.
- PD: This image is in the public domain. Dieses Bild ist gemeinfrei.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.

Bild	Autor	Lizenz
1		PD
2		PD
3		PD
4		PD
5		PD
6		PD
7		PD