Peter GOTTSCHLING

FORSCHUNG mit modernem C++

C++17-INTENSIVKURS für Wissenschaftler, Ingenieure und Programmierer

HANSER

Peter Gottschling

Forschung mit modernem C++

Peter Gottschling

Forschung mit modernem C++

Autor:

Dr. Peter Gottschling, Leipzig

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Internet: www.hanser-fachbuch.de

Lektorat: Dipl.-Ing. Natalia Silakova-Herzberg
Herstellung: Anne Kurth
Covergestaltung: Max Kostopoulos
Titelbild: © shutterstock.com/Immersion Imagery
Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München
Satz: Peter Gottschling
Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen
Printed in Germany

Print-ISBN 978-3-446-45846-8
E-Book-ISBN 978-3-446-45981-6

Meinen wundervollen Kindern Vincent, Daniel, Yanis und Anissa

Vorwort

"The world is built on C++ (and its C subset)."

– Herb Sutter

Die Infrastrukturen bei Google, Amazon und Facebook bestehen aus Komponenten und Diensten, die in C++ entworfen und implementiert wurden. Auch ein erheblicher Teil der Technologie von Betriebssystemen, Netzwerkgeräten und Speichersystemen ist mit C++ verwirklicht. In Telekommunikationssystemen werden fast alle Festnetz- und Mobilfunkverbindungen mit C++-Software aufgebaut. Und Schlüsselkomponenten in Industrie- und Transportsystemen – wie automatisierte Mautsysteme und autonome Fahrzeuge – basieren auf C++.

In Wissenschaft und Technik werden die meisten hochwertigen Softwarepakete heute in C++ realisiert. Die Stärke der Sprache beweist sich, wenn Projekte eine bestimmte Größe überschreiten und Datenstrukturen und Algorithmen nicht mehr trivial sind. Es ist kein Wunder, dass viele – wenn nicht die meisten – Simulationssoftwareprogramme in der Informatik heute in C++ realisiert werden: FLUENT, Abaqus, deal.II, FEniCS, OpenFOAM, G+SMO. Auch Embedded-Systeme werden dank leistungsfähigerer Embedded-Prozessoren und verbesserter Compiler zunehmend in C++ programmiert. Und die neuen Anwendungsbereiche Internet of Things (IoT) und Embedded Edge Intelligence werden alle von C++-Plattformen wie TensorFlow, Caffe2 und CNTK beherrscht.

Wesentliche, täglich von Ihnen genutzte Dienste basieren auf C++: Von Ihrem Mobiltelefon bis zu Ihrem Auto, in der Kommunikations- und Industrieinfrastruktur sowie wichtige Elemente in Medien- und Unterhaltungsdiensten enthalten alle C++-Komponenten. C++-Dienste und -Anwendungen sind in der modernen Gesellschaft allgegenwärtig. Der Grund ist einfach. Die Sprache C++ hat sich mit ihren Anforderungen weiterentwickelt und ist in vielerlei Hinsicht führend bei der Produktivität der Programmierung und der Ausführungseffizienz. Beide Charakteristika machen es zur bevorzugten Sprache für Anwendungen, die skalierbar sein müssen.

Gründe, C++ zu lernen

Wie keine andere Sprache meistert C++ das gesamte Spektrum von der hardwarenahen Programmierung auf der einen bis hin zur abstrakten High-Level-Programmierung auf der anderen Seite. Die Low-Level-Programmierung – wie nutzerdefinierbare Speicherverwaltung – ermöglicht es Ihnen als Programmierer zu verstehen, was wirklich während der Ausführung passiert. Dies hilft Ihnen wiederum, das Verhalten von Programmen in anderen Sprachen zu verstehen. In C++ können Sie extrem effiziente Programme schreiben, deren Performance der von in Maschinensprache geschriebenem Code nur geringfügig nachsteht, wobei letzteres ein Vielfaches an Entwicklungsaufwand erfordert.

Allerdings sollten Sie mit dem Hardcore-Performance-Tuning erst einmal warten und sich zunächst auf klare und aussagekräftige Software konzentrieren. Hier kommen die High-Level-Features von C++ ins Spiel. Die Sprache unterstützt eine Vielzahl von Programmierparadigmen direkt: objekt-orientierte Programmierung (Kapitel 6), generische (Kapitel 3), Meta-Programmierung (Kapitel 5), parallele Programmierung (Abschnitt 4.6), prozedurale (Abschnitt 1.5) und weitere.

Verschiedene Programmiertechniken – wie RAII (Abschnitt 2.4.2.1) und Expressions-Templates (Abschnitt 5.3) – wurden in und für C++ erfunden. Da die Sprache so ausdrucksstark ist, war es oft möglich, diese neuen Techniken zu etablieren, ohne die Sprache zu ändern. Und wer weiß, vielleicht erfinden Sie eines Tages auch eine neue Technik.

Gründe, dieses Buch zu lesen

Das Material dieses Buches wurde an echten Menschen getestet. Der Autor hielt drei Jahre lang die Vorlesung "C++ für Wissenschaftler". Die Studenten, meist aus dem Fachbereich Mathematik, sowie einige aus der Physik und den Ingenieurwissenschaften, kannten C++ teilweise vorher gar nicht und waren am Ende des Kurses in der Lage, fortgeschrittene Techniken wie Expressions-Templates (Abschnitt 5.3) zu implementieren.

Sie können dieses Buch in Ihrem eigenen Tempo lesen: Direkt zur Sache, indem Sie dem Hauptpfad folgen, oder ausführlicher, indem Sie zusätzliche Beispiele und Hintergrundinformationen in Anhang A lesen (wobei es auch dann noch recht intensiv ist).

Die Schöne und das Biest

C++-Programme können auf vielfältige Weise geschrieben werden. In diesem Buch führen wir Sie sanft zu den anspruchsvolleren Stilen. Dies erfordert die Verwendung von fortgeschrittenen Features, die zunächst einschüchternd wirken könnten, was sich aber geben wird, sobald Sie sich daran gewöhnt haben. Dabei ist die High-Level-Programmierung nicht nur in einem breiteren Spektrum einsetzbar, sondern in der Regel auch genauso effizient, gelegentlich sogar effizienter und natürlich deutlich besser lesbar als die Low-Level-Programmierung.

Wir geben Ihnen einen ersten Eindruck mit einem einfachen Beispiel: Gradientenabstieg mit konstanter Schrittweite. Das Prinzip ist extrem einfach: Wir berechnen den steilsten Abstieg von $f(x)$ mit seinem Gradienten $g(x)$ und folgen dieser Richtung mit Schritten fester Größe zum nächsten lokalen Minimum. Selbst der algorithmische Pseudocode ist so einfach wie diese Beschreibung:

Input : Startwert x, Schrittweite s, Abbruchkriterium ε, Funktion f, Gradient g
Output : Lokales Minimum x
do
  $x = x - s \cdot g(x)$
while $|\Delta f(x)| \geq \varepsilon$;

Algorithmus 1 : Gradientenabstiegsverfahren

Für diesen einfachen Algorithmus haben wir zwei recht unterschiedliche Implementierungen geschrieben. Schauen Sie doch einfach mal rein, ohne zu versuchen, die technischen Details zu verstehen.

```
void gradient_descent(double* x,
    double* y, double s, double eps,
    double(*f)(double, double),
    double(*gx)(double, double),
    double(*gy)(double, double))
{
    double val= f(*x, *y), delta;
    do {
        *x-= s * gx(*x, *y);
        *y-= s * gy(*x, *y);
        double new_val= f(*x, *y);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
}
```

```
template <typename Value, typename P1,
          typename P2, typename F,
          typename G>
Value gradient_descent(Value x, P1 s,
                       P2 eps, F f, G g)
{
    auto val= f(x), delta= val;
    do {
        x-= s * g(x);
        auto new_val= f(x);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
    return x;
}
```

Auf den ersten Blick sehen sich beide Version recht ähnlich, und wir werden Ihnen gleich sagen, welche wir bevorzugen. Die Erste ist im Prinzip reines C, d.h. auch mit einem C-Compiler kompilierbar. Ihr Vorteil ist, dass die berechnete Optimierung direkt sichtbar ist: eine 2D-Funktion mit `double`-Werten (dargestellt durch die hervorgehobenen Funktionsparameter). Wir bevorzugen die zweite Version, da sie allgemeiner anwendbar ist: Funktionen beliebiger Dimension mit beliebigen Werttypen, (erkennbar an den markierten Typen und Funktionsparametern). Überraschenderweise ist die allgemeinere Umsetzung nicht weniger effizient. Im Gegenteil, die als `F` und `G` übergebenen Funktionen können `inline` verwendet werden (siehe Abschnitt 1.5.3), so dass der Overhead des Funktionsaufrufs eingespart wird, während die explizite Verwendung von (unschönen) Funktionszeigern in der linken Version diese Optimierung erschwert oder komplett unmöglich macht.

Ein längeres Beispiel für den Vergleich von altem und neuem Stil finden Sie in Anhang A.1 (für den wirklich geduldigen Leser). Dort manifestiert sich der Nutzen der modernen Programmierung viel deutlicher als in dem hier gezeigten Einführungsbeispiel. Aber wir wollen Sie nicht zu lange mit dem Vorgeplänkel aufhalten.

Sprachen in Wissenschaft und Technik

> *"Es wäre schön, wenn jede Art von numerischer Software ohne Effizienzverlust in C++ geschrieben werden könnte; aber wenn dies nur durch Kompromittieren des C++-Typsystems möglich wäre, sollten wir uns lieber auf Fortran, Assembler oder architekturspezifische Erweiterungen verlassen."*
>
> *– Bjarne Stroustrup*

Wissenschaftliche und technische Software wird in verschiedenen Sprachen geschrieben, und welche sich am Besten geeignet, hängt von den Zielen und verfügbaren Ressourcen ab:

- Mathematische Werkzeuge wie MATLAB, Mathematica oder R sind ausgezeichnet, wenn wir ihre vorhandenen Algorithmen verwenden können. Wenn wir unsere eigenen Algorithmen mit feingranularen (z.B. skalaren) Operationen implementieren, werden wir einen

deutlichen Leistungsabfall verzeichnen. Dies ist noch nicht kritisch, wenn die Probleme klein sind oder der Nutzer eine unendliche Geduld aufbringt; andernfalls sollten wir über alternative Sprachen nachdenken.

- Python eignet sich hervorragend für die schnelle Software-Entwicklung und enthält bereits wissenschaftliche Bibliotheken wie "SciPy" und "NumPy," Oft sind diese Bibliotheken in C oder C++ implementiert und daher auch einigermaßen effizient. Auch hier gilt: nutzerdefinierte Algorithmen mit vielen feingranularen Operationen führen zu einer deutlichen Leistungseinbuße. Python ist hervorragend, um kleine und mittlere Aufgaben effizient zu lösen. Wenn Projekte ausreichend groß werden, wird es immer wichtiger, dass der Compiler strenger wird (z.B. dass er Zuweisungen abgelehnt, wenn die Argumenttypen nicht passen).
- Fortran ist auch großartig, wenn wir bestehende, sorgfältig optimierte Operationen – wie dichte Matrixmultiplikation – nutzen können. Es ist auch bestens geeignet, um die Hausaufgaben von alten Professoren zu erledigen (wenn sie nur nach dem fragen, was in Fortran einfach ist). Die Einführung neuer Datenstrukturen ist nach den Erfahrungen des Autors recht umständlich, und das Schreiben eines großen Simulationsprogramms in Fortran ist eine ziemliche Herausforderung – heute nur noch freiwillig von einer schwindenden Minderheit in Angriff genommen.
- C ermöglicht eine gute Performance, und eine große Menge an Software ist in C geschrieben. Die Kernsprache ist relativ klein und leicht zu erlernen. Die Herausforderung besteht darin, große und fehlerfreie Software mit den einfachen und riskanten Sprachfeatures, insbesondere Zeiger (Abschnitt 1.8.2) und Makros (Abschnitt 1.9.2.1), zu erstellen. Der letzte Standard wurde 2011 veröffentlicht, daher der Name C11. Die meisten seiner Funktionen – aber nicht alle – sind seit C++14 auch in C++ enthalten.
- Sprachen wie Java, C# und PHP sind wahrscheinlich eine gute Wahl, wenn die Hauptkomponente der Anwendung eine Web- oder Grafikschnittstelle ist und nicht zu viele Berechnungen durchgeführt werden.
- C++ brilliert besonders, wenn wir große, hochwertige Software mit guter Performance entwickeln. Dennoch muss der Entwicklungsprozess nicht langsam und schmerzhaft sein. Mit den richtigen Abstraktionen können wir unsere C++-Programme sehr schnell entwickeln. Wir sind optimistisch, dass in Zukunft noch viele wissenschaftliche Bibliotheken entstehen werden.

Je mehr Sprachen wir kennen, desto mehr Auswahl haben wir natürlich. Je besser wir diese Sprachen beherrschen, desto fundierter wird unsere Auswahl sein. Zudem enthalten große Projekte oft Komponenten in verschiedenen Sprachen, wobei in den meisten Fällen zumindest die leistungskritischen Kerne in C oder C++ realisiert sind. Alles in allem ist das Lernen von C++ eine faszinierende Reise, und ein tiefes Verständnis davon wird Sie auf jeden Fall zu einem großartigen Programmierer machen.

Typographische Konventionen

Neue Termini werden *"kursiv in Anführungszeichen"* dargestellt. Wichtige Begriffe werden *kursiv* gesetzt.

Wichtige Hinweise werden in einer Box mit Pfeil gegeben. Wenn es sich um Tipps zum Programmieren handelt, sollten Sie nur aus sehr gewichtigen Gründen dagegen verstoßen.

Boxen mit Ausrufungszeichen enthalten Regeln, die unbedingt befolgt werden sollten.

C++-Quellen sind `blau und nichtproportional` gedruckt. Wichtige Programmdetails werden durch **`fette Schrift`** hervorgehoben. Innerhalb von Programmen sind Klassen, Funktionen, Variablen und Konstanten kleingeschrieben und können optional Unterstriche enthalten (snake_case). Eine Ausnahme bilden Matrizen, die in der Regel mit einem Großbuchstaben benannt werden. Template-Parameter beginnen mit einem Großbuchstaben und können weitere enthalten (CamelCase). Programmausgaben und Kommandozeilenbefehle sind in der gleichen nichtproportionalen Schrift gesetzt, jedoch in schwarz.

Programme, die C++11, C++14 oder C++17-Funktionen erfordern, sind mit entsprechenden Randboxen markiert. Einige Programme, die nur wenige C++11-Features verwenden, welche einfach durch C++03-Ausdrücke ersetzt werden können, sind nicht immer explizit gekennzeichnet.

⇒ `verzeichnis/quell_code.cpp`

Bis auf sehr kurze Code-Illustrationen wurden alle Programmierbeispiele in diesem Buch auf drei Compilern getestet: `g++`, `clang++` und Visual Studio. Die Dateinamen mit Pfad innerhalb des Repos sind für die getesteten Programmbeispiele, welche für das betreffende Thema relevant sind, am Anfang des entsprechenden Absatzes oder Abschnitts gekennzeichnet.

Alle Programme sind in einem öffentlichen Repository auf GitHub – *https://github.com/petergottschling/dmc2* – verfügbar und können mit dem folgenden Befehl geklont werden:

```
git clone https://github.com/petergottschling/dmc2.git
```

Unter Windows ist es praktischer, TortoiseGit zu verwenden; siehe *tortoisegit.org*.

Da wir aus eigener Erfahrung wissen, dass jede Redundanz die Gefahr von Inkonsistenz in sich birgt, haben wir die Programmbeispiele für die 2. englische Auflage und die 1. deutsche Ausgabe in einem gemeinsamen Repository bereitgestellt. In der deutschen Version sind die meisten Kommentare und Ausgaben nach dem Kopieren ins Buch übersetzt worden, aber sonst unterscheiden sich die Programmschnipsel im Buch nicht von den Quellen auf GitHub.

Danksagungen

Chronologisch beginnend, möchte ich Karl Meerbergen und seinen Kollegen für den ersten 80-seitigen Text danken, der 2008 als Blockvorlesung an der KU Leuven von Karl und mir gehalten wurde. Im Laufe der Zeit wurden die meisten Passagen neu geschrieben, aber das Originaldokument lieferte den ersten und unentbehrlichen Impuls, der für den gesamten Schreibprozess maßgeblich war. Ich stehe in Mario Mulanskys tiefer Schuld für seinen Beitrag des Abschnitts 7.1 über die Implementierung von ODE-Lösern.

Unendlich dankbar bin ich Jan Christiaan van Winkel und Fabio Fracassi, die jedes noch so kleine Detail des Manuskripts überprüft und viele Vorschläge zur Standardkonformität und Verständlichkeit gemacht haben.

Besonders danken möchte ich auch Bjarne Stroustrup für seine strategischen Tipps zur Gestaltung des Buches, den Kontakt zu Addison-Wesley, die großzügige Wiederverwendung seines gut aufbereiteten Materials und (nicht zu vergessen) die Erschaffung von C++. All diese Korrekturleser drängten mich zum Glück, das alte Vorlesungsmaterial so weit wie möglich mit den Features von C++11 und C++14 zu aktualisieren.

Darüber hinaus danke ich Karsten Ahnert für seine Empfehlungen und Markus Abel für seine Hilfe, die Langatmigkeit aus dem Vorwort zu beseitigen. Theodore Omtzigt gebührt Dank dafür, die Motivation im Vorwort für die zweite englische und die erste deutsche Auflage noch fokussierter formuliert zu haben. Als ich nach einer interessanten Anwendung von Zufallszahlen für Abschnitt 4.2.2.6 suchte, schlug Jan Rudl die Kursentwicklung von Aktienportfolios aus seiner Vorlesung vor.

Ich bin der Technischen Universität Dresden verbunden, die mich drei Jahre lang Vorlesungen über dieses Thema im Fachbereich Mathematik halten ließ, und ich schätze das konstruktive Feedback der Studenten aus der Lehrveranstaltung. Ebenso danke ich den Teilnehmern meiner C++-Seminare für viele Ideen.

Ich bin meinem Verleger Greg Doench zutiefst dankbar, dass er in diesem Buch meinen teils ernsthaften, teils unbeschwerten Stil akzeptiert hat. Dafür, dass er lange über strategische Entscheidungen diskutiert hat, bis wir beide zufrieden waren, und mir professionelle Unterstützung geboten hat, ohne die das Buch nie veröffentlicht worden wäre. Elizabeth Ryan gebührt Dank für das Management des gesamten Produktionsprozesses.

Last but not least danke ich meinen Kindern Yanis, Anissa, Vincent und Daniel von ganzem Herzen dafür, dass sie so viel von unserer gemeinsamen Zeit geopfert haben, damit ich an dem Buch arbeiten konnte.

Danksagungen zur deutschen Ausgabe

Ich bin dem Carl Hanser Verlag sehr dankbar, dass er dieses Buch nun auf Deutsch veröffentlicht und mich dabei in allen Bereichen unterstützt hat. Besonders dankbar bin ich Natalia Silakova, die den gesamten Veröffentlichungsprozess organisiert hat.

Stephan Korell möchte ich für die Unterstützung in LaTeX-Fragen danken. Meiner Kollegin Katja Mülsow bin ich wahnsinnig dankbar, dass sie alle LaTeX-Befehle in den unformatierten deutschen Text eingefügt hat und das Buch aus sprachlicher Sicht Korrektur gelesen hat.

Über den Autor

Peter Gottschlings berufliche Leidenschaft ist das Entwickeln wissenschaftlicher Spitzensoftware, und er hofft, viele Leser mit diesem Virus infizieren zu können. Diese Berufung führte zur Entstehung der Matrix Template Library 4 und zum Mitverfassen anderer Bibliotheken, einschließlich der Boost Graph Library. Diese Programmiererfahrungen wurden in mehreren C++-Kursen an Universitäten und in professionellen Trainingsseminaren geteilt, die schließlich zu diesem Buch führten.

Er ist Mitglied des ISO C++-Standardkomitees, stellvertretender Obmann des deutschen Normenausschusses für Programmiersprachen und Gründer der C++-User-Group in Dresden. In seinen jungen und wilden Jahren an der TU Dresden studierte er parallel Informatik und Mathematik bis zum Vordiplom und schloss ersteres mit einer Promotion ab. Nach einer Odyssee durch akademische Einrichtungen gründete er seine eigene Firma SimuNova und kehrte vor einigen Jahren in seine Heimatstadt Leipzig zurück.

Inhalt

1 Grundlagen

"An meine Kinder:
Macht euch nie darüber lustig, mir mit Computerkram helfen zu müssen.
Ich habe euch beigebracht, wie man einen Löffel benutzt."

– Sue Fitzmaurice

Im ersten Kapitel werden wir die grundlegenden Features von C++ einführen. Wie im gesamten Buch werden wir sie aus verschiedenen Blickwinkeln betrachten, aber nicht versuchen, jedes denkbare Detail herauszuarbeiten (was ohnehin nicht machbar ist). Für detailliertere Fragen zu bestimmten Features empfehlen wir die Online-Handbücher, wie *http://en.cppreference.com* und *http://www.cplusplus.com/*.

1.1 Unser erstes Programm

⇒ c++03/hello42.cpp

Als Einführung in die Sprache C++ sehen wir uns das folgende Beispiel an:

```
#include <iostream>

int main ()
{
    std::cout << "Die ultimative Antwort auf die Frage nach dem Leben,\n"
              << "dem Universum und dem ganzen Rest ist:"
              << std::endl << 6 * 7 << std::endl;
    return 0;
}
```

Dies ergibt laut Douglas Adams [2]:

```
Die ultimative Antwort auf die Frage nach dem Leben,
dem Universum und dem ganzen Rest ist:
42
```

Dieses kurze Beispiel illustriert bereits mehrere Features von C++:

- Ein- und Ausgabe sind nicht Teil der Kernsprache, sondern werden von der Standard-Bibliothek bereitgestellt. Sie müssen explizit *"eingebunden"* werden. Sonst können unsere Programme weder lesen noch schreiben.
- Die Standard-I/O hat ein *"Stream-Modell"* und heißt daher `<iostream>`. Um seine Funktionalität zu nutzen, haben wir sie in der ersten Zeile mit `#include` eingebunden (inkludiert).
- Jedes C++-Programm beginnt mit dem Aufruf der Hauptfunktion `main`. Sie gibt einen ganzzahligen Wert zurück, wobei 0 für ein erfolgreiches Ende steht.

- Geschweifte Klammern (engl. braces) markieren einen *"Anweisungsblock"* – auch zusammengesetzte Anweisung, engl. compound statement, genannt.
- `std::cout` und `std::endl` sind in `<iostream>` definiert. Ersteres ist ein Ausgabestrom, der es uns ermöglicht, Text auf dem Bildschirm zu schreiben. `std::endl` beendet eine Zeile. Wir können eine Zeile auch mit dem Sonderzeichen `\n` beenden.
- Der Operator `<<` kann verwendet werden, um Objekte an einen Ausgabe-Stream wie `std::cout` zu übergeben und somit eine Ausgabeoperation durchzuführen. Bitte beachten Sie, dass der Operator in Programmen mit zwei kleiner-als-Zeichen (`<<`) geschrieben wird. Für ein eleganteres Druckbild verwenden wir stattdessen ein (doppeltes) französisches Guillemet in einem einzigen Symbol.
- `std::` bedeutet, dass der Typ oder die Funktion aus dem Standard-*"Namensraum"* (engl. `namespace`) verwendet wird. Namensräume helfen uns, unsere Namen zu organisieren und mit Namenskonflikten zu umgehen; siehe Abschnitt 3.2.1.
- Zeichenkettenkonstanten (genauer gesagt Literale) werden in doppelte Anführungszeichen gesetzt. Im Buch verwenden wir hauptsächlich den englischen Begriff *"String"*.
- Der Ausdruck `6 * 7` wird ausgewertet und als ganze Zahl an `std::cout` übergeben. In C++ hat jeder Ausdruck einen Typ. Manchmal müssen wir als Programmierer den Typ explizit deklarieren und andere Male kann der Compiler ihn für uns ermitteln. 6 und 7 sind literale Konstanten vom Typ `int` und dementsprechend ist auch ihr Produkt `int`.

Bevor Sie weiterlesen, empfehlen wir Ihnen dringend, dass Sie dieses kleine Programm auf Ihrem Computer übersetzen (kompilieren). Sobald es kompiliert und läuft, können Sie ein wenig damit spielen, z.B. weitere Operationen und Ausgaben hinzufügen. Und gegebenenfalls die Fehlermeldungen betrachten. Schließlich ist der einzige Weg, eine Sprache wirklich zu lernen, sie zu benutzen, selbst wenn am Anfang mehr schiefgeht als klappt. Wenn Sie bereits wissen, wie man einen Compiler oder sogar eine C++-IDE benutzt, können Sie den Rest dieses Abschnitts überspringen.

Linux: Jede Distribution liefert zumindest den GNU C++-Compiler – üblicherweise schon installiert (siehe das kurze Intro in Abschnitt B.1). Wenn wir unser Programm `hello42.cpp` aufrufen wollen, ist dies ganz einfach mit dem Befehl:

```
g++ hello42.cpp
```

Einer obskuren Tradition des letzten Jahrhunderts folgend, wird die daraus resultierende Binärdatei standardmäßig "`a.out`" genannt. Spätestens wenn wir mehrere Programme in einem Verzeichnis haben, werden wir der Binärdatei einen aussagekräftigeren Namen geben wollen:

```
g++ hello42.cpp -o hello42
```

Wir können auch das Build-Tool `make` verwenden, das Standardregeln für die Erstellung von Binärdateien hat (Übersicht in Abschnitt 7.2.2.1). Wir müssen nur:

```
make hello42
```

aufrufen und `make` sucht im aktuellen Verzeichnis nach einer ähnlich benannten Programmquelle. Es wird "`hello42.cpp`" finden und den standardmäßigen C++-Compiler des Systems aufgerufen, da "`.cpp`" eine Standarddateiendung für C++-Quellen ist. Sobald wir unser Programm kompiliert haben, können wir es auf der Kommandozeile als aufrufen:

```
./hello42
```

Unsere Binärdatei kann ohne weitere Software ausgeführt werden, und wir können sie auf ein anderes kompatibles Linux-System[1] kopieren und dort lassen laufen.

Windows: Wenn Sie MinGW nutzen, können Sie Ihre Programme auf die gleiche Weise kompilieren wie unter Linux. Verwenden Sie Visual Studio, müssen Sie zuerst ein Projekt erstellen. Zu Beginn ist es das Einfachste, die Projektvorlage für eine Konsolenanwendung zu nutzen, wie bspw. unter *http://www.cplusplus.com/doc/tutorial/introduction/visualstudio* beschrieben. Wenn Sie das Programm ausführen, haben Sie bei bestimmten Konfigurationen nur ein paar Millisekunden Zeit, um die Ausgabe zu lesen, bevor die Konsole geschlossen wird. Um die Lesezeit auf eine Sekunde zu verlängern, können Sie einfach den nicht-portablen Befehl `Sleep(1000)` einfügen (und `<windows.h>` einbinden). Mit C++11 oder höher kann die Wartephase portabel programmiert werden:

```
std::this_thread::sleep_for(std::chrono::seconds(1));
```

Binden Sie dafür `<chrono>` und `<thread>` ein. Microsoft bietet kostenlose Versionen von Visual Studio namens "Express", die die Standardsprache genau so gut unterstützen wie ihre professionellen Äquivalente. Der Unterschied besteht darin, dass die professionellen Editionen mehr Entwickler-Bibliotheken (SDKs) enthalten. Da diese im Buch nicht verwendet werden, können Sie die "Express"-Version zum Testen unserer Beispiele nutzen.

IDE: Kurze Programme können auch mit einem gewöhnlichen Editor geschrieben werden. Vor allem bei größeren Projekten ist es ratsam, eine *"Integrierte Entwicklungsumgebung"* (engl. *"Integrated Development Environment"* , kurz IDE) zu verwenden. Damit kann man sehen, wo eine Funktion definiert oder verwendet wird, die Dokumentation innerhalb des Codes anzeigen, Namen projektweit suchen oder ersetzen und vieles andere mehr. KDevelop ist eine freie, in C++ geschriebene IDE aus der KDE-Community. Es ist vermutlich die effizienteste IDE unter Linux und sowohl `git` als auch `CMake` sind bereits integriert. Eclipse ist in Java geschrieben und merklich langsamer (soll aber besser geworden sein). Trotzdem sind viele Entwickler damit recht produktiv. Visual Studio ist eine sehr solide IDE, die eine produktive Entwicklung unter Windows ermöglicht und in neueren Versionen auch eine Integration von `CMake`-Projekten bietet. Die produktivste Umgebung für sich zu finden, benötigt etwas Zeit und Experimentiererei. Außerdem es ist natürlich abhängig vom persönlichen und kollaborativen Geschmack, der sich im Laufe der Zeit auch weiterentwickeln kann.

1.2 Variablen

C++ ist eine stark typisierte Sprache, im Gegensatz zu vielen Skriptsprachen. Das bedeutet, dass jede Variable einen Typ hat und sich dieser nie ändert. Eine Variable wird durch eine Anweisung deklariert, die mit dem Typ beginnt, gefolgt von einem Variablennamen und einer optionalen Initialisierung:

[1] Oft ist die Standardbibliothek dynamisch verlinkt (vgl. Abschnitt 7.2.1.4) und dann ist ihr Vorhandensein in der gleichen Version auf dem anderen System Teil der Kompatibilitätsanforderungen.

```
int    i1= 2;              // Ausrichtung nur für Lesbarkeit
int    i2, i3= 5;
float  pi= 3.14159;
double x= -1.5e6;          // -1500000
double y= -1.5e-6;         // -0.0000015
char   c1= 'a', c2= 35;
bool   cmp= i1 < pi,       // -> true
       happy= true;
```

Die beiden Schrägstriche `//` sind der Anfang eines einzeiligen Kommentars, d.h. alles ab den doppelten Schrägstrichen bis zum Ende der Zeile wird ignoriert. Im Prinzip ist das alles, was man über Kommentare wirklich wissen muss. Nichtsdestotrotz werden wir in Abschnitt 1.9.1 noch ein bisschen mehr darüber sagen.

1.2.1 Fundamentale Typen

Die grundlegendsten Typen (engl. intrinsic types) in C++ sind die in Tabelle 1.1 aufgeführten fundamentalen Typen. Sie sind Teil der Kernsprache und immer verfügbar.

Tabelle 1.1 Grundlegende Typen

Name	Semantik
`char`	Buchstabe oder sehr kleine Zahl
`short`	recht kleine ganze Zahl
`int`	reguläre ganze Zahl
`long`	große ganze Zahl
`long long`	sehr große ganze Zahl
`unsigned`	vorzeichenlose Versionen der vorangegangenen
`signed`	vorzeichenbehaftete Versionen der vorangegangenen
`float`	Gleitkommazahl mit einfacher Genauigkeit
`double`	Gleitkommazahl mit doppelter Genauigkeit
`long double`	Gleitkommazahl mit mehr als doppelter Genauigkeit
`bool`	logischer Typ

Die ersten fünf Typen sind ganze Zahlen mit nicht abnehmender Länge. Zum Beispiel ist `int` mindestens so lang wie `short`, d.h. es ist normalerweise länger, aber nicht zwangsläufig. Die genaue Länge jedes Typs ist von der Implementierung abhängig; z.B. kann `int` 16, 32 oder 64 Bit sein. Alle diese Typen können als `signed` (vorzeichenbehaftet) oder `unsigned` (vorzeichenlos) gekennzeichnet werden. Ersteres hat keinen Einfluss auf Integer-Zahlen (außer `char`), da sie standardmäßig `signed` sind.

Wenn wir einen ganzzahligen Typ als `unsigned` deklarieren, haben wir keine negativen Werte, dafür jedoch doppelt so viele positive (plus eins, wenn wir Null als weder positiv noch negativ betrachten). `signed` und `unsigned` können als Adjektive für das Substantiv `int` betrachtet werden, wenn das Adjektiv allein verwendet wird. Gleiches gilt auch für die Adjektive `short`, `long` und `long long`.

Der Typ `char` kann auf zwei Arten verwendet werden: für Buchstaben und recht kleine Zahlen. Abgesehen von wirklich exotischen Architekturen hat der Typ fast immer eine Länge von 8 Bit. So können wir entweder Werte von -128 bis 127 (`signed`) oder von 0 bis 255 (`unsigned`) darstellen und alle numerischen Operationen mit ihnen durchführen, die für ganze Zahlen verfügbar sind. Wenn weder `signed` noch `unsigned` deklariert wird, hängt es von der Implementierung des Compilers ab, was verwendet wird. Die Verwendung von `char` oder `unsigned char` für kleine Zahlen kann jedoch nützlich sein, wenn es sehr viele davon gibt.

Logische Werte werden am besten als `bool` dargestellt. Eine boolsche Variable kann die Werte `true` und `false` annehmen.

Die Eigenschaft der nicht abnehmenden Länge gilt in gleicher Weise für Gleitkommazahlen: `float` ist kürzer oder gleich lang wie `double`, was wiederum kürzer oder gleich lang wie `long double` ist. Typische Größen sind 32 Bit für `float`, 64 Bit für `double` und 80 Bit für `long double`.

1.2.2 Characters und Strings

Wie bereits erwähnt, kann der Typ `char` verwendet werden, um Zeichen zu speichern:

```
char c= 'f';
```

Wir können auch jeden Buchstaben darstellen, dessen Code in 8 Bit passt. Es kann sogar mit Zahlen gemischt werden; z.B. führt `'a' + 7` in der Regel zu `'h'`, abhängig von der zugrundeliegenden Kodierung der Buchstaben. Wir raten dringend davon ab, damit zu spielen, da die mögliche Verwirrung wahrscheinlich zu unnötiger Zeitverschwendung führt.

Von C haben wir die Möglichkeit geerbt, Zeichenketten als Arrays von `char` darzustellen.

```
char name[9]= "Herbert";
```

Diese alten C-Strings enden alle mit einer binären `0` als `char`-Wert. Fehlt die `0`, laufen die Algorithmen bis zum nächsten Speicherplatz mit einem `0`-Byte weiter. Eine andere große Gefahr besteht beim Anhängen von Zeichenketten: `name` hat keinen zusätzlichen Platz, zusätzliche Zeichen überschreiben irgendwelche andere Daten. Alle String-Operationen richtig zu implementieren, ohne den Speicher zu beschädigen oder längere Strings abzuschneiden, ist bei diesen alten Zeichenketten alles andere als trivial. Wir empfehlen daher dringend, sie nur für literale Werte zu verwenden.

Der C++-Compiler unterscheidet zwischen einfachen und doppelten Anführungszeichen: `'a'` ist das Zeichen “a” (es hat den Typ `char`) und `"a"` ist ein Array mit einer binären `0` als Abschluss (d.h. sein Typ ist `const char[2]`).

Im Gegensatz dazu erlaubt die Klasse `string` aus der Bibliothek `<string>` einen viel einfacheren und zugleich sichereren Umgang mit Zeichenketten:

```
#include <string>

int main()
{
    std::string name= "Herbert";
}
```

C++-Strings verwenden dynamischen Speicher und verwalten ihn selbst. Wenn wir also mehr Text an einen String anhängen, müssen wir uns keine Sorgen über Speicherzugriffsfehler oder das Abschneiden von Strings machen:

```
name= name + ", unser cooler Antiheld"; // Mehr dazu später
```

Viele aktuelle Implementierungen verwenden auch eine Optimierung für kurze Strings (z.B. bis 16 Byte), die direkt im `string`-Objekt gespeichert werden statt zusätzlichen Speicher anzufordern. Diese Optimierung kann die aufwendige Speicherallokation und -freigabe deutlich reduzieren.

C++14 Da Text in doppelten Anführungszeichen als `char`-Array interpretiert wird, benötigen wir die Möglichkeit, einen Text als String zu bezeichnen. Dies geschieht mit dem Nachsatz `s`, z.B. `"Herbert"s`. Leider hat es bis C++14 gedauert, um dies zu ermöglichen. Eine explizite Konvertierung wie `string("Herbert")` war schon immer möglich. In C++17 wurde eine leichtgewichtige, konstante Sicht auf Strings hinzugefügt, die wir in Abschnitt 4.4.5 vorstellen werden.

1.2.3 Variablen deklarieren

Deklarieren Sie Variablen so spät wie möglich, in der Regel direkt vor der ersten Verwendung und wenn möglich nicht bevor Sie sie initialisieren können.

Dies macht Programme besser lesbar, wenn sie lang werden. Es erlaubt dem Compiler auch, den Speicher mit verschachtelten Bereichen effizienter zu nutzen.

C++11 Wir können auch den Compiler den Typ einer Variablen für uns ermitteln lassen, z.B:

```
auto i4= i3 + 7;
```

Der Typ von `i4` ist der gleiche wie der von `i3 + 7`, nämlich `int`. Auch automatisch ermittelte Typen ändern sich nicht wieder, und was auch immer `i4` noch zugewiesen wird, wird in `int` konvertiert. Wir werden später sehen, wie nützlich `auto` in der fortgeschrittenen Programmierung ist. Für einfache Variablendeklarationen (wie die in diesem Abschnitt) ist es normalerweise besser, den Typ explizit zu deklarieren. `auto` wird in Abschnitt 3.4 ausführlich behandelt.

1.2.4 Konstanten

Syntaktisch gesehen sind Konstanten wie spezielle Variablen in C++ mit dem zusätzlichen Attribut `const`:

```
const int    ci1= 2;
const int    ci3;              // Fehler: kein Wert
const float  pi= 3.14159;
const char   cc= 'a';
const bool   cmp= ci1 < pi;
```

Da diese Objekte nicht geändert werden können, ist es zwingend erforderlich, ihre Werte in der Deklaration festzulegen. Die zweite Konstantendeklaration verletzt diese Regel, und der Compiler wird ein solches Fehlverhalten nicht tolerieren.

Konstanten können überall dort verwendet werden, wo Variablen erlaubt sind, solange sie nicht verändert werden. Andererseits sind Konstanten wie im obigen Beispiel normalerweise bereits während der Kompilierung bekannt. Dies ermöglicht viele Arten von Optimierungen, und die Konstanten können sogar als Argumente von Typen verwendet werden (wir werden darauf später in Abschnitt 5.1.4 zurückkommen).

1.2.5 Literale

Literale wie `2` oder `3.14` werden ebenfalls typisiert. Einfach ausgedrückt, werden ganze Zahlen je nach Anzahl der Stellen als `int`, `long` oder `unsigned long` behandelt. Jede Zahl mit einem Punkt oder einem Exponenten (z.B. `3e12` $\equiv 3 \cdot 10^{12}$) wird als `double` angesehen.

Literale anderer Typen erhalten wir, wenn wir einer Endung aus Tabelle 1.2 hinzufügen:

Tabelle 1.2 Typen von Literalen

Literal	Typ
`2`	`int`
`2u`	`unsigned`
`2l`	`long`
`2ul`	`unsigned long`
`2.0`	`double`
`2.0f`	`float`
`2.0l`	`long double`

In den meisten Fällen ist es nicht notwendig, die Typen der Literale explizit zu deklarieren, da die implizite Konvertierung in gemischten Ausdrücken (engl. coercion) zwischen fundamentalen numerischen Typen normalerweise unseren Erwartungen entspricht.

Es gibt jedoch drei Gründe, warum wir auf den Typ der Literale achten sollten:

Verfügbarkeit: Wenn wir eine Funktion aufrufen wollen, die es für `int` bzw. `double` nicht gibt und der Wert auch nicht implizit in einen Typen, für den es eine Funktion gibt, umgewandelt werden kann. Beispielsweise stellt die Standardbibliothek eine Klasse für komplexe Zahlen zur Verfügung, bei dem der Typ für Real- und Imaginärteil vom Anwender parametrisiert werden kann:

```
std::complex<float> z(1.3, 2.4), z2;
```

Leider werden Operationen nur zwischen dem Typ selbst und dem zugrundeliegenden realen Typ angeboten (und Argumente werden hier nicht konvertiert).[2] Folglich können wir `z` nicht mit einem `int` oder `double`, sondern nur mit `float` multiplizieren:

```
z2= 2 * z;        // Fehler: kein int * complex<float>
z2= 2.0 * z;      // Fehler: kein double * complex<float>
z2= 2.0f * z;     // Okay:  float * complex<float>
```

[2] Gemischte Arithmetik ist jedoch realisierbar, wie in [16] demonstriert wurde.

Mehrdeutigkeit: Wenn eine Funktion für verschiedene Argumenttypen überladen ist (Abschnitt 1.5.4), kann ein Argument wie `0` mehrdeutig sein, während für ein qualifiziertes Argument wie `0u` eine eindeutige Überladung existieren kann.

Genauigkeit: Das Problem der Genauigkeit tritt auf, wenn wir mit `long double` arbeiten. Da ein nicht-qualifiziertes Literal den Typ `double` hat, können wir Ziffern verlieren, bevor wir es einer `long double`-Variable zuweisen:

```
long double third1= 0.3333333333333333333;  // könnte Ziffern verlieren
long double third2= 0.33333333333333333331; // exakt
```

Ganzzahlige Literale, die mit einer Null beginnen, werden als Oktalzahlen interpretiert, z.B:

```
int o1= 042;          // int o1= 34;
int o2= 084;          // Fehler: keine 8 und 9 in Oktalzahlen
```

Hexadezimale Literale können durch Voranstellen von `0x` oder `0X` geschrieben werden:

```
int h1= 0x42;         // int h1= 66;
int h2= 0xfa;         // int h2= 250;
```

C++14 Mit dem Präfix `0b` or `0B` können wir nun auch binäre Literale schreiben:

```
int b1= 0b11111010;   // int b1= 250;
```

C++14 Um lange Literale lesbarer zu gestalten, können wir die Ziffern durch Apostrophe trennen:

```
long               d=   6'546'687'616'861'129l;
unsigned long      ulx= 0x139'ae3b'2ab0'94f3;
int                b=   0b101'1001'0011'1010'1101'1010'0001;
const long double  pi=  3.141'592'653'589'793'238'462l;
```

C++17 Gleitkomma-Literale können jetzt auch hexadezimal geschrieben werden:

```
auto f1= 0x10.1p0f; // 16.0625
auto d2= 0x1ffp10;  // 523264
```

Für sie ist der Exponent obligatorisch – daher brauchten wir im ersten Beispiel `p0`. Durch das Suffix `f` ist `f1` ein `float`, der den Wert $16^1 + 16^{-1} = 16.0625$ speichert. Diese Literale verwenden drei Basen: Die Pseudo-Mantisse ist hexadezimal, skaliert mit Potenzen zur Basis 2, wobei der Exponent als Dezimalzahl angegeben wird. Somit hat `d2` den Wert $511 \times 2^{10} = 523264$. Hexadezimale Literale wirken anfangs zwar etwas seltsam, aber sie erlauben es uns, binäre Gleitkommazahlen ohne Rundungsfehler zu deklarieren.

String-Literale werden als Arrays von `char` geschrieben:

```
char s1[]= "Alter C-Stil"; // lieber nicht
```

Allerdings sind diese Arrays alles andere als nutzerfreundlich und wir fahren besser mit `string`-Objekten, die direkt mit einem String-Literal initialisiert werden können:

```
#include <string>

std::string s2= "In C++ lieber so.";
```

Sehr langer Text kann in mehrere Teilstrings aufgeteilt werden:

```
std::string s3= "Das ist langer und langatmiger Text, "
                "der nicht ganz auf eine Zeile passt.";
```

Für weitere Details zu Literalen siehe [49, §6.2].

1.2.6 Werterhaltende Initialisierung

C++11

Angenommen, wir initialisieren eine `long`-Variable mit einer großen Zahl:

```
long l2= 1234567890123;
```

Dies kompiliert problemlos und funktioniert korrekt – wenn `long` wie auf den meisten 64-Bit-Plattformen 64 Bit nutzt. Wenn `long` nur 32 Bit lang ist (z.B. in Visual Studio oder beim Kompilieren mit dem Flag `-m32`), ist der obige Wert zu lang. Das Programm kompiliert aber trotzdem (eventuell mit einer Warnung) und läuft mit einem anderen Wert, z.B. indem die führenden Bits abgeschnitten werden.

C++11 führt eine *"werterhaltende"* Initialisierung ein, bei der die Genauigkeit nicht verloren geht oder wie es der Standard nennt, dass die Werte nicht *"verengt"* werden; siehe Abschnitt C.2.4 für die genaue Definition. Dies wird mit der vereinheitlichten Initialisierung erreicht, die wir hier einführen möchten und in Abschnitt 2.3.4 weiter vertiefen werden. Werte in geschweiften Klammern dürfen nicht verengt werden:

```
long l= {1234567890123};
```

Nun prüft der Compiler, ob die Variable `l` den Wert auf der Zielplattform speichern kann. Diese Kontrolle des Compilers bewahrt uns auch vor Genauigkeitsverlusten bei der Initialisierung. Eine gewöhnliche Initialisierung eines `int` durch eine Gleitkommazahl ist dagegen aufgrund der impliziten Konvertierung erlaubt:

```
int i1= 3.14;        // kompiliert trotz Verengung (unser Risiko)
int i1n= {3.14};     // Verengungsfehler: nach  Komma abgeschnitten
```

Die neue Initialisierungsform in der zweiten Zeile verbietet dies, da sie den Nachkommateil der Gleitkommazahl abschneiden würde. Ebenso wird die Zuweisung negativer Werte an vorzeichenlose Variablen oder Konstanten bei der traditionellen Initialisierung toleriert, aber in der neuen Form verworfen:

```
unsigned u2= -3;     // kompiliert trotz Verengung (unser Risiko)
unsigned u2n= {-3};  // Fehler durch Verengung: keine negativen Werte
```

In den vorherigen Beispielen haben wir literale Werte in den Initialisierungen verwendet und der Compiler prüft, ob ein bestimmter Wert mit diesem Typ darstellbar ist:

```
float f1= {3.14};    // okay
```

Nun, der Wert 3.14 kann in keinem binären Fließkommaformat absolut genau dargestellt werden, aber der Compiler kann `f1` auf den Wert setzen, der 3.14 am nächsten kommt. Wenn eine `float`-Variable mit einer `double`-Variablen (keinem Literal oder Konstante) initialisiert wird, müssen alle möglichen `double`-Werte berücksichtigt werden und ob sie alle verlustfrei in `float` konvertierbar sind:

```
double d;
...
float f2= {d};        // Fehler durch Verengung
```

Beachten Sie, dass die Verengung zwischen zwei Typen wechselseitig sein kann:

```
unsigned u3= {3};
int      i2= {2};

unsigned u4= {i2};    // Fehler: keine negativen Werte
int      i3= {u3};    // Fehler: nicht alle großen Werte
```

Die Typen `signed int` und `unsigned int` haben die gleiche Größe, aber nicht alle Werte des einen Typs sind im jeweils anderen darstellbar.

1.2.7 Gültigkeitsbereiche

Gültigkeitsbereiche (engl. Scopes) bestimmen die Lebensdauer und Sichtbarkeit von (nichtstatischen) Variablen und Konstanten und etablieren eine Struktur in unseren Programmen.

1.2.7.1 Globale Definitionen

Jede Variable, die wir in einem Programm verwenden wollen, muss mit ihrer Typangabe zuvor im Code deklariert worden sein. Eine Variable kann sowohl im globalen als auch im lokalen Bereich liegen. Eine globale Variable wird außerhalb aller Funktionen deklariert. Nach ihrer Deklaration können globale Variablen von überall im Code genutzt werden, auch innerhalb von Funktionen. Das klingt zunächst sehr praktisch, weil die Variablen leicht verfügbar sind, aber wenn unsere Software wächst, wird es schwieriger und schmerzhafter, alle Änderungen von globalen Variablen nachzuvollziehen. Jede Code-Änderung birgt irgendwann das Potenzial, eine ganze Lawine von Fehlern auszulösen.

Verwenden Sie keine globalen Variablen!

Sie können im gesamten Programm verwendet werden und daher ist es extrem mühsam, den Überblick über ihre Verwendung zu wahren.

Globale Konstanten wie:

```
const double pi= 3.14159265358979323846264338327950288419716939;
```

sind in Ordnung, da sie keine Seiteneffekte verursachen können.

1.2.7.2 Lokale Definitionen

Eine lokale Variable wird innerhalb des Körpers einer Funktion deklariert. Ihre Sicht- und Verfügbarkeit ist auf den in `{ }` eingeschlossenen Block ihrer Deklaration beschränkt. Genauer gesagt, beginnt der Scope einer Variablen mit ihrer Deklaration und endet mit der schließenden Klammer des Deklarationsblocks.

Wenn wir `pi` in der Funktion `main` definieren:

```
int main ()
{
    const double pi= 3.14159265358979323846264338327950288419716939;
    std::cout << "pi ist " << pi << ".\n";
}
```

dann existiert sie nur dort. Wir können Blöcke innerhalb von Funktionen und innerhalb anderer Blöcke definieren:

```
int main ()
{
    {
        const double pi= 3.14159265358979323846264338327950288419716939;
    }
    std::cout << "pi ist " << pi << ".\n"; // Fehler: pi nicht im Scope
}
```

In diesem Beispiel ist die Definition von pi auf den Block innerhalb der Funktion beschränkt. Daher ist eine Ausgabe im Rest der Funktion ein Fehler:

```
»pi« is not defined in this scope.
```

weil der Zugriff auf pi außerhalb dessen Gültigkeitsbereichs ist.

1.2.7.3 Verbergen

Wenn eine Variable mit dem gleichen Namen in geschachtelten Bereichen existiert, dann ist jeweils nur eine Variable sichtbar. Die Variable im inneren Bereich verbirgt die gleichnamigen in den äußeren Bereichen, zum Beispiel:

```
int main ()
{
    int a= 5;          // definiert a#1
    {
        a= 3;          // ändert a#1, a#2 noch nicht definiert
        int a;         // definiert a#2
        a= 8;          // ändert a#2, a#1 ist verborgen
        {
            a= 7;      // ändert a#2
        }
    }                  // Ende der Gültigkeit von a#2
    a= 11;             // ändert a#1 (a#2 aus dem Scope)

    return 0;
}
```

Durch das Ausblenden müssen wir die Lebensdauer und die Sichtbarkeit von Variablen unterscheiden. Beispielsweise lebt a#1 von seiner Deklaration bis zum Ende der Hauptfunktion. Sie ist jedoch nur von ihrer Deklaration bis zur Deklaration von a#2 und wieder nach dem Schließen des Blocks mit a#2 sichtbar. Im Prinzip ist die Sichtbarkeit die Lebensdauer minus der Zeit, in der sie verborgen ist. Es ist ein Fehler, denselben Variablennamen zweimal im gleichen Bereich zu definieren.

Der Vorteil von Scopes ist, dass wir uns keine Gedanken darüber machen müssen, ob eine Variable bereits irgendwo außerhalb des Scopes definiert wurde. Sie wird dann verborgen und erzeugt dadurch keinen Namenskonflikt.[3] Leider macht das Verbergen die gleichnamigen Variablen im äußeren Bereich unzugänglich. Das können wir zum Teil mit einer cleveren Umbenennung bewältigen. Eine bessere Lösung zur Verwaltung von Verschachtelung und Zugreifbarkeit sind Namensräume, siehe Abschnitt 3.2.1.

`static`-Variablen sind die Ausnahme, die die Regel bestätigen: Sie leben bis zum Ende der Ausführung, sind aber nur in ihrem Bereich (Scope) sichtbar. Wir befürchten, dass ihre ausführliche Einführung zum jetzigen Zeitpunkt eher störend als hilfreich ist und haben die Diskussion in Abschnitt A.2.1 verschoben.

1.3 Operatoren

C++ ist reich an eingebauten Operatoren:

- Berechnungen:
  - Arithmetik: `++`, `+`, `*`, `%`, ...
  - Boolesche Operationen:
    - Vergleiche: `<=`, `!=`, ...
    - Logik: `&&` und `||`
  - Bitweise: `~`, `≪`, `≫`, `&`, `^`, und `|`
- Zuweisungen: `=`, `+=`, ...
- Programmfluss: Funktionsaufruf, `?:` und `,`
- Speicherbehandlung: `new` und `delete`
- Datenzugriff: `.`, `->`, `[ ]`, `*`, ...
- Typbehandlung: `dynamic_cast`, `typeid`, `sizeof`, `alignof`, ...
- Fehlerbehandlung: `throw`

Dieser Abschnitt gibt einen Überblick über die Operatoren. Für einige Operatoren ist es sinnvoll, zunächst die relevanten Sprachmerkmale einzuführen; z.B. ergibt die Bereichsauflösung erst mit einem Verständnis für Namensräume einen Sinn. Die meisten Operatoren können für Nutzertypen überschrieben werden, d.h. wir können entscheiden, welche Berechnungen durchgeführt werden, wenn Argumente unserer Typen in Ausdrücken vorkommen.

Am Ende dieses Abschnitts (Tabelle 1.9) finden Sie eine Übersichtstabelle der Operatorprioritäten. Es kann nicht schaden, diese Seite auszudrucken oder zu kopieren und sie neben den Monitor zu hängen; viele Leute tun dies und fast niemand kennt die gesamte Prioritätenliste auswendig. Auch sollten Sie nicht zögern, Klammern um Teilausdrücke zu setzen, wenn Sie sich über die Prioritäten unsicher sind oder wenn Sie glauben, dass es für andere Programmierer, die mit Ihren Quellen arbeiten, verständlicher ist. Wenn Sie Ihren Compiler bitten, pedantisch zu sein, nimmt er diesen Job oft zu ernst und fordert Sie auf, jede Menge überflüssige

[3] Im Gegensatz zu Makros, ein gnadenloses Altlasten-Feature von C, das um jeden Preis vermieden werden sollte, weil es alle Struktur und Zuverlässigkeit der Sprache untergräbt.

Klammern hinzuzufügen, weil er Sie mit den Vorrangregeln überfordert glaubt. Im Anhang C, Tabelle C.1 finden Sie eine vollständige Liste aller Operatoren mit Kurzbeschreibungen und Referenzen.

1.3.1 Arithmetische Operatoren

Tabelle 1.3 listet die in C++ verfügbaren arithmetischen Operatoren auf. Wir haben sie nach ihren Prioritäten sortiert, aber schauen wir sie uns einzeln an.

Tabelle 1.3 Arithmetische Operatoren

Operation	Ausdruck
Post-Inkrement	`x++`
Post-Dekrement	`x--`
Pre-Inkrement	`++x`
Pre-Dekrement	`--x`
Unäres Plus	`+x`
Unäres Minus	`-x`
Multiplikation	`x * y`
Division	`x / y`
Modulo	`x % y`
Addition	`x + y`
Subtraktion	`x - y`

Die ersten Operationen sind In- und Dekrementieren. Diese können verwendet werden, um eine Zahl um 1 zu erhöhen oder zu verringern. Da sie den Wert der Zahl ändern, sind sie nur für Variablen und nicht für Zwischenergebnisse sinnvoll:

```
int i= 3;
i++;              // i ist jetzt 4
const int j= 5;
j++;              // Fehler: j ist konstant
(3 + 5)++;        // Fehler: 3 + 5 nur Zwischenergebnis
```

Kurz gesagt, die In- und Dekrementoperationen benötigen ein Objekt, das modifizierbar und adressierbar ist. Der Fachbegriff für ein adressierbares Datenelement ist l-Wert (siehe Definition C.1 im Anhang C). In unserem obigen Code-Schnipsel gilt dies nur für `i`, da `j` konstant und `3 + 5` nicht adressierbar ist.

Beide Notationen – Präfix und Postfix – wirken sich auf eine Variable aus, indem sie 1 addieren oder subtrahieren. Der Wert eines Ausdrucks ist jedoch für Präfix- und Postfix-Operatoren unterschiedlich: Die Präfix-Operatoren geben den geänderten Wert zurück und Postfix den alten:

```
int i= 3, j= 3;
int k= ++i + 4;   // i ist 4, k ist 8
int l= j++ + 4;   // j ist 4, l ist 7
```

Am Ende haben sowohl `i` als auch `j` den Wert 4. Bei der Berechnung von `l` wurde jedoch der alte Wert von `j` verwendet, während die erste Addition den bereits erhöhten Wert von `i` nutzte.

Generell ist es besser, auf die Verwendung von In- und Dekrement in mathematischen Ausdrücken zu verzichten und diese durch `j+1` und dergleichen zu ersetzen oder die Erhöhungen und Verminderungen separat durchzuführen. Es ist für den menschlichen Leser einfacher zu verstehen und für den Compiler besser zu optimieren, wenn mathematische Ausdrücke keine Seiteneffekte haben. Wir werden bald sehen, warum (Abschnitt 1.3.12).

Das unäre Minus negiert den Wert einer Zahl:

```
int i= 3;
int j= -i;          // j ist -3
```

Das unäre Plus hat keine arithmetische Wirkung auf Standardtypen. Für Nutzertypen können wir das Verhalten von unärem Plus und Minus definieren. Wie in Tabelle 1.3 dargestellt, haben diese unären Operatoren die gleiche Priorität wie Pre-Inkrement und Pre-Dekrement.

Die Operationen `*` und `/` sind natürlich Multiplikation und Division, und beide sind auf allen numerischen Typen definiert. Wenn beide Argumente in einer Division ganze Zahlen sind, dann wird der Nachkommateil des Ergebnisses abgeschnitten (Rundung gegen Null). Der Operator `%` liefert den Rest der ganzzahligen Division. Daher sollten beide Argumente einen ganzzahligen Typ haben.

Last but not least, die Operatoren `+` und `-` zwischen zwei Variablen oder Ausdrücken symbolisieren Addition und Subtraktion.

Die semantischen Details der Operationen – wie Ergebnisse gerundet werden oder wie Überlauf behandelt wird – sind in der Sprache nicht spezifiziert. Aus Performancegründen überlassen C++-Compiler dies wenn möglich der zugrundeliegenden Hardware.

Generell haben unäre Operatoren eine höhere Priorität als binäre. In den seltenen Fällen, in denen sowohl Postfix- als auch Präfix-Notationen verwendet wurden, wird Postfix priorisiert.

Unter den binären Operatoren haben wir das gleiche Verhalten, das wir aus der Mathematik kennen: Multiplikation und Division gehen Addition und Subtraktion voraus und die Operationen sind links-assoziativ, d.h.:

```
x - y + z
```

wird immer ausgewertet als:

```
(x - y) + z
```

Was der Standard allerdings nicht festlegt und dessen sich viele nicht bewusst sind, ist die Reihenfolge der Argumentauswertung bei Funktionen und Operatoren, zum Beispiel:

```
int i= 3, j= 7, k;
k= f(++i) + g(++i) + j;
```

In diesem Beispiel garantiert die Assoziativität, dass die erste Addition vor der zweiten ausgeführt wird. Ob aber zuerst der Ausdruck `f(++i)` oder `g(++i)` berechnet wird, hängt von der Compiler-Implementierung ab. So kann `k` entweder `f(4) + g(5) + 7` oder `f(5) + g(4) + 7` sein, sogar `f(5) + g(5) + 7`. Außerdem können wir nicht davon ausgehen, dass das Ergebnis auf einer anderen Plattform gleich ist. Prinzipiell ist es gefährlich, Werte innerhalb von Ausdrücken zu verändern. Dies funktioniert unter bestimmten Umständen, aber wir müssen es

immer wieder testen und dem große Aufmerksamkeit schenken. Insgesamt ist es sinnvoller, ein paar Zeichen mehr einzutippen und die Änderungen separat vorzunehmen. Mehr zu diesem Thema in Abschnitt 1.3.12.

⇒ c++03/num_1.cpp

Mit diesen Operatoren können wir unser erstes numerisches Programm schreiben:

```
#include <iostream>

int main ()
{
    const float r1= 3.5, r2 = 7.3, pi = 3.14159;

    float area1 = pi * r1*r1;
    std::cout ≪ "Ein Kreis mit Radius " ≪ r1 ≪ " hat die Fläche "
              ≪ area1 ≪ "." ≪ std::endl;

    std::cout ≪ "Der Durchschnitt von " ≪ r1 ≪ " und " ≪ r2
              ≪ " ist " ≪ (r1 + r2) / 2 ≪ "." ≪ std::endl;
}
```

Wenn die Argumente einer binären Operation unterschiedliche Typen haben, werden ein oder beide Argumente automatisch in einen gemeinsamen Typ gemäß den Regeln in Abschnitt C.2 konvertiert.

Die Konvertierung kann zu einem Genauigkeitsverlust führen. Gleitkommazahlen werden ganzzahligen Zahlen vorgezogen, und offensichtlich führt die Umwandlung eines 64-Bit-`long` in einen 32-Bit-`float` zu einem Genauigkeitsverlust; selbst ein 32-Bit-`int` kann nicht immer korrekt als 32-Bit-`float` dargestellt werden, da einige Bits für den Exponenten benötigt werden. Es gibt auch Fälle, in denen die Zielvariable das korrekte Ergebnis darstellen könnte, die Genauigkeit aber bereits in den Zwischenberechnungen verloren geht. Folgendes Beispiel illustriert dies:

```
long l= 1234567890123;
long l2= l + 1.0f - 1.0;    // ungenau
long l3= l + (1.0f - 1.0); // korrekt
```

Dies ergab auf dem Rechner des Autors:

```
l2 = 1234567954431
l3 = 1234567890123
```

Bei `l2` verlieren wir durch die Zwischenumrechnungen an Genauigkeit, während `l3` korrekt berechnet wurde. Dies ist zwar ein künstliches Beispiel, aber Sie sollten sich der Gefahr ungenauer Zwischenergebnisse bewusst sein. Gerade bei großen Berechnungen müssen die numerischen Algorithmen sorgfältig ausgewählt werden, damit sich die Fehler nicht hochschaukeln. Die Frage der Ungenauigkeit wird uns im nächsten Abschnitt glücklicherweise nicht stören.

1.3.2 Boolesche Operatoren

Boolesche Operatoren sind logische und relationale Operatoren. Beide geben `bool`-Werte zurück, wie der Name schon sagt. Diese Operatoren und ihre Bedeutung sind in Tabelle 1.4 nach Priorität gruppiert aufgeführt.

Tabelle 1.4 Boolesche Operatoren

Operation	Ausdruck
Negation	`!b`
Größer als	`x > y`
Größer-gleich	`x >= y`
Kleiner als	`x < y`
Kleiner-gleich	`x <= y`
Gleich	`x == y`
Ungleich	`x != y`
Logisches UND	`b && c`
Logisches ODER	`b || c`

Binäre relationale und logische Operatoren haben niedrigere Prioritäten als alle arithmetischen. Das bedeutet, dass der Ausdruck `4 >= 1 + 7` als `4 >= (1 + 7)` ausgewertet wird.

Verwenden Sie immer `bool` für logische Ausdrücke.

Bitte beachten Sie, dass Vergleiche nicht so verkettet werden können:

```
bool in_bound= min <= x <= y <= max;      // Fehler
```

Stattdessen brauchen wir die etwas längere logische Reduktion:

```
bool in_bound= min <= x && x <= y && y <= max;
```

Im folgenden Abschnitt werden wir ganz ähnliche Operatoren sehen.

1.3.3 Bitweise Operatoren

Diese Operatoren erlauben es uns, einzelne Bits von ganzzahligen Typen zu testen oder zu manipulieren. Sie sind wichtig für die Systemprogrammierung, aber weniger für die moderne Anwendungsentwicklung. Tabelle 1.5 listet alle Operatoren nach Priorität auf.

Tabelle 1.5 Bitweise Operatoren

Name	Semantik
Komplement	`~x`
Linksverschiebung	`x ≪ y`
Rechtsverschiebung	`x ≫ y`
Bitweises UND	`x & y`
Bitweises exklusives ODER	`x ^ y`
Bitweises inklusives ODER	`x | y`

Die Operation `x << y` verschiebt die Bits von `x` um `y` Positionen nach links und `x >> y` entsprechend nach rechts.[4] In den meisten Fällen werden Nullen hineingeschoben, mit Ausnahme von negativen vorzeichenbehafteten Werten in einer Rechtsverschiebung, wo das Verhalten implementierungsabhängig ist.

Das bitweise UND kann verwendet werden, um ein bestimmtes Bit eines Wertes zu testen. Das bitweise inklusive ODER kann ein Bit setzen und das exklusive ODER kippen. Obwohl diese Operationen in wissenschaftlichen Anwendungen weniger wichtig sind, werden wir sie in Abschnitt 3.6.1 zum algorithmischen Entertainment verwenden.

1.3.4 Zuweisung

Der Wert eines Objektes (veränderlicher l-Wert) kann durch eine Zuweisung gesetzt werden:

```
object= expr;
```

Wenn die Typen nicht übereinstimmen, wird `expr` nach Möglichkeit in den Objekttyp konvertiert. Die Zuweisung ist rechts-assoziativ, so dass ein Wert nacheinander mehreren Objekten in einem Ausdruck zugewiesen werden kann:

```
o3= o2= o1= expr;
```

Apropos Zuweisung: Der Autor möchte an dieser Stelle erklären, warum er das Symbol linksbündig ausrichtet. Die meisten binären Operatoren sind in dem Sinne symmetrisch, dass beide Argumente Werte sind. Im Gegensatz dazu muss bei einer Zuweisung auf der linken Seite eine veränderliche Variable stehen und auf rechten kann sich ein beliebiger Ausdruck (eines passenden Typs) befinden. Diese Asymmetrie der Argumente spiegelt sich in anderen Sprachen in asymmetrischen Symbolen wieder (z.B. `:=` in Pascal), während der Autor in C++ stattdessen einen asymmetrischen Abstand verwendet.

Die zusammengesetzten Zuweisungen wenden eine arithmetische oder bitweise Operation auf das Objekt auf der linken Seite mit dem Argument auf der rechten Seite an; beispielsweise sind die folgenden beiden Operationen äquivalent:

```
a+= b;          // Entspricht Folgendem:
a= a + b;
```

Alle Zuweisungsoperatoren haben eine niedrigere Priorität als jede arithmetische oder bitweise Operation, so dass der rechte Ausdruck immer vor der zusammengesetzten Zuweisung ausgewertet wird:

```
a*= b + c;      // Entspricht Folgendem:
a= a * (b + c);
```

Die Zuweisungsoperatoren sind in Tabelle 1.6 aufgeführt. Sie sind alle rechts-assoziativ und haben die gleiche Priorität.

[4] Auch für diese Operationen komprimieren wir die doppelten Kleiner-als- und Größer-als-Symbole zu französischen Guillemets für einen schöneren Druck.

Tabelle 1.6 Zuweisungsoperatoren

Operation	Ausdruck
Einfache Zuweisung	`x= y`
Multipliziere und weise zu	`x*= y`
Dividiere und weise zu	`x/= y`
Modulo und weise zu	`x%= y`
Addiere und weise zu	`x+= y`
Subtrahiere und weise zu	`x-= y`
Linksverschiebung und Zuweisung	`x<<= y`
Rechtsverschiebung und Zuweisung	`x>>= y`
UND und Zuweisung	`x&= y`
Inklusives ODER und Zuweisung	`x|= y`
Exklusive ODER und Zuweisung	`x^= y`

1.3.5 Programmablauf

Es gibt drei Operatoren, die den Programmablauf steuern. Zunächst wird ein Funktionsaufruf in C++ wie ein Operator behandelt. Eine detaillierte Beschreibung von Funktionen und ihrer Aufrufe finden Sie in Abschnitt 1.5.

Der Bedingungsoperator `c ? x : y` wertet die Bedingung `c` aus, und wenn sie erfüllt ist, hat der Ausdruck den Wert `x`, ansonsten `y`. Er kann als Alternative zu Verzweigungen mit `if` verwendet werden, insbesondere an Stellen, an denen nur ein Ausdruck und keine Anweisung erlaubt ist; siehe Abschnitt 1.4.3.1.

Ein ganz besonderer Operator in C++ ist der Komma-Operator, der eine sequentielle Auswertung ermöglicht. Seine Semantik ist die Auswertung der beiden Teilausdrücke von links nach rechts. Der Wert des gesamten Ausdrucks ist der des rechten Teilausdrucks:

```
3 + 4, 7 * 9.3
```

Das Ergebnis des Ausdrucks ist 65.1 und die Berechnung des ersten Teilausdrucks ist in diesem Fall völlig irrelevant. Die Teilausdrücke können auch den Komma-Operator enthalten, so dass beliebig lange Sequenzen definiert werden können. Mit Hilfe des Komma-Operators können mehrere Ausdrücke an Programmstellen ausgewertet werden, an denen nur eine Anweisung erlaubt ist. Ein typisches Beispiel ist das Inkrementieren mehrerer Indizes in einer `for`-Schleife (Abschnitt 1.4.4.2):

```
++i, ++j
```

Bei Verwendung als Funktionsargument benötigt der Kommaausdruck umschließende Klammern, ansonsten wird das Komma als Trennung von Funktionsargumenten interpretiert.

1.3.6 Speicherverwaltung

Die Operatoren `new` und `delete` allokieren bzw. deallokieren Speicher. Wir verschieben ihre Beschreibung auf Abschnitt 1.8.2, da es nicht sinnvoll ist, darüber zu reden, ohne vorher Zeiger eingeführt zu haben.

1.3.7 Zugriffsoperatoren

C++ bietet mehrere Operatoren für den Zugriff auf Teilstrukturen, für das Verweisen – d.h. das Ermitteln der Adresse einer Variablen oder Funktion – und das Dereferenzieren – d.h. das Zugreifen auf den Speicher bei gegebener Adresse (z.B. in einem Zeiger). Sie sind in Tabelle 1.7 aufgeführt. Wir werden nach der Einführung von Zeigern und Klassen in Abschnitt 2.2.3 demonstrieren, wie man Zugriffsoperatoren verwendet.

Tabelle 1.7 Zugriffsoperatoren

Operation	Ausdruck	Referenz
Member-Zugriff	`x.m`	Abschnitt 2.2.3
Dereferenzierter Member-Zugriff	`p->m`	Abschnitt 2.2.3
Indizierter Zugriff	`x[i]`	Abschnitt 1.8.1
Dereferenzieren	`*x`	Abschnitt 1.8.2
Member dereferenzieren	`x.*q`	Abschnitt 2.2.3
Dereferenzierten Member dereferenzieren	`p->*q`	Abschnitt 2.2.3

1.3.8 Typbehandlung

Die Operatoren für den Umgang mit Typen werden in Kapitel 5 vorgestellt, wenn wir Meta-Programme schreiben, die mit Typen arbeiten. Für den Moment möchten wir uns darauf beschränken, sie in Tabelle 1.8 aufzulisten.

Tabelle 1.8 Operatoren zur Typbehandlung

Operation	Ausdruck
Laufzeittypbestimmung	`typeid(x)`
Bestimmung eines Typs	`typeid(t)`
Größe eines Objektes	`sizeof(x)` or `sizeof x`
Größe eines Typs	`sizeof(t)`
Anzahl der Objektargumente	`sizeof...(p)`
Anzahl der Typargumente	`sizeof...(P)`
Ausrichtung eines Objektes	`alignof(x)`
Ausrichtung eines Typs	`alignof(t)`

C++11

Beachten Sie, dass der `sizeof`-Operator bei Verwendung auf einem Ausdruck der einzige ist, der ohne Klammern anwendbar ist. `alignof` wurde in C++11 eingeführt; alle anderen existieren bereits in C++03.

1.3.9 Fehlerbehandlung

Der `throw`-Operator wird verwendet, um eine Ausnahme in der Ausführung anzuzeigen (z.B. unzureichender Speicher); siehe Abschnitt 1.6.2.

1.3.10 Überladung

Ein sehr mächtiger Aspekt von C++ ist, dass der Programmierer Operatoren für neue Typen definieren kann. Dies wird in Abschnitt 2.7 erläutert. Operatoren der eingebauten Typen können nicht geändert werden. Wir können jedoch definieren, wie eingebaute Typen mit neuen Typen interagieren, d.h. wir können gemischte Operationen – wie die Multiplikation eines `double` mit einer Matrix – überladen.

Die meisten Operatoren können überladen werden. Ausnahmen sind:

`::`	Angabe des Scopes;
`.`	Member-Auswahl (vielleicht in C++20);
`.*`	Member-Auswahl über Zeiger;
`?:`	Bedingungsoperator;
`sizeof`	Größe eines Typs oder Objektes;
`sizeof...`	Argumentzahl;
`alignof`	Speicherausrichtung eines Objektes oder Typs und
`typeid`	Typidentifikator.

Das Überladen von Operatoren in C++ gibt uns viel Freiheit und wir müssen mit dieser Freiheit weise umgehen. Wir kommen im nächsten Kapitel auf dieses Thema zurück, wenn wir die Operatoren tatsächlich überladen (warten Sie bitte bis Abschnitt 2.7).

1.3.11 Operatorprioritäten

Tabelle 1.9 gibt einen kompakten Überblick über die Operatorprioritäten. Aus Platzgründen haben wir Notationen für Typen und Ausdrücke (z.B. `typeid`) kombiniert und die verschiedenen Notationen für `new` und `delete` zusammengelegt. Das Symbol `@=` steht für alle Operationen wie `+=`, `-=` etc. Eine detailliertere Zusammenfassung der Operatoren mit Semantik finden Sie in Anhang C, Tabelle C.1.

Tabelle 1.9 Operatorprioritäten

Operatorprioritäten			
class`::`*member*	*nspace*`::`*member*	`::`*name*	`::`*qualified-name*
object`.`*member*	*pointer*`->`*member*	*expr*`[`*expr*`]`	*expr*(*expr_list*)
type(*expr_list*)	*lvalue*`++`	*lvalue*`--`	`typeid`(*type/expr*)
`*_cast<`*type*`>` (*expr*)			
`sizeof` *expr*	`sizeof`(*type*)	`sizeof...`(*pack*)	`alignof` (*type/expr*)
`++`*lvalue*	`--`*lvalue*	~*expr*	`!`*expr*
`-`*expr*	`+`*expr*	`&`*lvalue*	`*`*expr*
`new` ... *type*...	`delete` []$_{opt}$ *pointer*	(*type*) *expr*	
object`.*` *member_ptr*	*pointer*`->*` *member_ptr*		
expr `*` *expr*	*expr* `/` *expr*	*expr* % *expr*	
expr `+` *expr*	*expr* `-` *expr*		
expr `<<` *expr*	*expr*`>>` *expr*		
expr `<` *expr*	*expr* `<=` *expr*	*expr* `>` *expr*	*expr* `>=` *expr*
expr `==` *expr*	*expr* `!=` *expr*		
expr `&` *expr*			
expr `^` *expr*			
expr `|` *expr*			
expr `&&` *expr*			
expr `||` *expr*			
expr `?` *expr*`:` *expr*			
lvalue `=` *expr*	*lvalue* `@=` *expr*		
`throw` *expr*			
expr , *expr*			

1.3.12 Vermeiden Sie Seiteneffekte!

"Wahnsinn: Immer wieder dasselbe tun und andere Ergebnisse erwarten."

– Unbekannt[5]

Bei Programmen mit Seiteneffekten ist es nicht verrückt, ein anderes Ergebnis für den gleichen Input zu erwarten. Im Gegenteil, es ist sehr schwierig, das Verhalten eines Programms, dessen

[5] Irrtümlicherweise Albert Einstein, Benjamin Franklin und Mark Twain zugeschrieben. Es wird in "Sudden Death" von Rita Mae Brown zitiert, aber die ursprüngliche Quelle scheint unbekannt zu sein. Anscheinend ist das Zitat selbst mit einem gewissen Wahnsinn behaftet.

Komponenten massiv interferieren, vorherzusagen. Außerdem ist es wahrscheinlich besser, ein deterministisches Programm mit dem falschen Ergebnis zu haben, als eines, das gelegentlich das richtige Ergebnis liefert, da letzteres normalerweise viel schwieriger zu debuggen ist.

In der C-Standardbibliothek gibt es die Funktion `strcpy` zum Kopieren von Strings. Die Funktion nimmt Zeiger auf das erste Zeichen der Quelle und des Ziels und kopiert die folgenden Zeichen, bis sie eine binäre Null findet. Dies kann mit einer einzigen Schleife realisiert werden, die sogar einen leeren Körper hat und die Kopie sowie die Inkrement-Operationen als Nebeneffekte des Fortsetzungstests durchführt:

```
while (*tgt++= *src++) ;
```

Sieht gruselig aus? Nun, irgendwie ist es das auch. Es ist jedoch absolut legaler C++-Code, obwohl einige Compiler im pedantischen Modus meckern könnten. Es ist eine gute mentale Übung, einige Zeit damit zu verbringen, über Operator-Prioritäten, Typen von Teilausdrücken und die Auswertungsreihenfolge nachzudenken.

Denken wir an etwas Einfacheres: Wir weisen dem `i`-ten Eintrag eines Arrays den Wert `i` zu und erhöhen den Wert `i` für die nächste Iteration:

```
v[i]= i++;
```

Kein Problem, sollte man denken. Ist es aber: Das Verhalten dieses Ausdrucks ist undefiniert. Warum? Das Post-Inkrement von `i` garantiert, dass wir danach den alten Wert von `i` zuweisen und danach inkrementieren. Dieses Inkrement kann vor der Auswertung des Ausdrucks `v[i]` ausgeführt werden, so dass wir möglicherweise `i` in `v[i+1]` schreiben.

Das letzte Beispiel soll demonstrieren, dass Nebenwirkungen nicht immer auf den ersten Blick erkennbar sind. Einige ziemlich knifflige Dinge könnten funktionieren, aber viel einfachere Dinge nicht. Schlimmer noch, ein Code könnte für eine Weile funktionieren, bis wir ihn auf einem anderen Compiler kompilieren oder eine neue Version Ihres Compilers die entsprechenden Implementierungsdetails ändert.

Die Schleife zum Kopieren von C-Strings ist ein Beispiel für exzellente Programmierkenntnisse und zeigt, dass die Prioritäten der Operatoren sinnvoll sind – es wurden keine Klammern benötigt. Dennoch ist ein solcher Programmierstil für modernes C++ nicht geeignet. Das Bestreben, den Code so weit wie möglich zu verkürzen, geht auf die Zeit des frühen C zurück, als das Tippen noch mühsamer war, mit Schreibmaschinen, die eher mechanisch als elektrisch waren, und Kartenstanzmaschinen ohne Monitor. Mit der heutigen Technologie sollte es für die Digital Natives kein Problem sein, einige zusätzliche Zeichen einzugeben.

Ein weiterer unschöner Aspekt der knappen Kopierimplementierung ist die Vermischung verschiedener Belange: Testen, Modifizieren und Traversieren. Ein wichtiges Konzept in der Softwareentwicklung ist die Trennung der Belange (engl. separation of concerns). Es trägt dazu bei, die Flexibilität zu erhöhen und die Komplexität zu verringern. In diesem Fall wollen wir die Komplexität der mentalen Prozesse reduzieren, die zum Verständnis der Umsetzung notwendig sind. Die Anwendung des Prinzips auf den berüchtigten Kopiereinzeiler könnte ergeben:

```
for (; *src; tgt++, src++)
    *tgt= *src;
*tgt= *src; // 0 am Ende kopieren
```

Nun können wir die drei Anliegen klar unterscheiden:

- Endtest: `*src`
- Modifikation: `*tgt= *src;`
- Traversierung: `tgt++, src++`

Es ist auch offensichtlicher, dass das Inkrementieren auf den Zeigern und das Testen und Zuweisen auf ihren referenzierten Inhalten durchgeführt wird. Die Implementierung ist nicht mehr so kompakt wie bisher, aber es ist viel einfacher, die Richtigkeit zu überprüfen. Es ist auch ratsam, den Nicht-Null-Test deutlicher zum Ausdruck zu bringen (`*src != 0`).

Es gibt eine Klasse von Programmiersprachen – die *"funktionalen"*, in denen Werte nicht geändert werden können. C++ zählt offensichtlich nicht dazu. Dessen ungeachtet tun wir uns selbst einen großen Gefallen, wenn wir so viel wie möglich in einem funktionalen Stil programmieren. Wenn wir zum Beispiel eine Zuweisung schreiben, sollte sich nur die Variable links vom Zuweisungssymbol ändern. Dazu müssen wir die Veränderung durch einen konstanten Ausdruck ersetzen: z.B. `++i` durch `i+1`. Ein rechtsseitiger Ausdruck ohne Nebenwirkungen hilft uns, das Programmverhalten besser zu verstehen und erleichtert es dem Compiler, den Code zu optimieren. Als Faustregel gilt: Verständlichere Programme haben ein deutlich höheres Optimierungspotenzial.

1.4 Ausdrücke und Anweisungen

C++ unterscheidet zwischen Ausdrücken und Anweisungen (engl. statement). Salopp formuliert könnte man sagen, dass jeder Ausdruck zu einer Anweisung wird, wenn man ein Semikolon anhängt. Wir möchten dieses Thema jedoch etwas ausführlicher diskutieren.

1.4.1 Ausdrücke

Lassen Sie es uns rekursiv von unten nach oben konstruieren. Jeder Variablenname (wie `x`, `y`), jede Konstante und jedes Literal ist ein Ausdruck. Ein oder mehrere Ausdrücke, die mit einem Operator kombiniert werden, bilden wiederum einen Ausdruck, z.B. `x + y` oder `x * y + z`. In vielen Sprachen ist die Zuweisung eine Anweisung. In C++ ist sie ein Ausdruck, beispielsweise `x= y + z`. Folglich kann sie auch innerhalb anderer Zuweisung wiederverwendet werden: `x2= x= y + z`. Zuweisungen werden von rechts nach links ausgewertet. Ein- und Ausgabeoperationen wie:

```
std::cout ≪ "x ist " ≪ x ≪ "\n"
```

sind ebenfalls Ausdrücke. Ein Funktionsaufruf mit Ausdrücken als Argument ist ein Ausdruck, z.B `abs(x)` oder `abs(x * y + z)`. Daher können Funktionsaufrufe geschachtelt werden: `pow(abs(x), y)`. Beachten Sie, dass eine Verschachtelung nicht möglich wäre, wenn es sich bei den Funktionsaufrufen um Anweisungen handeln würde.

Da die Zuweisung ein Ausdruck ist, kann sie als Argument einer Funktion verwendet werden, genauso wie I/O-Operationen:

```
print(cout ≪ "x = " ≪ x ≪ "\n", x= y, "Ich bin so ein Nerd!");
```

Überflüssig zu erwähnen, dass dies nicht sonderlich lesbar ist und mehr Verwirrung stiftet als alles andere. Ein von Klammern umgebener Ausdruck ist ebenfalls ein Ausdruck, z.B. `(x + y)`. Da die Gruppierung mit Klammern vor allen Operatoren kommt, können wir die Reihenfolge der Auswertung nach Bedarf ändern: `x * (y + z)` berechnet die Addition zuerst.

1.4.2 Anweisungen

Jeder der obigen Ausdrücke gefolgt von einem Semikolon ist eine Anweisung, z.B:

```
x= y + z;
y= f(x + z) * 3.5;
```

Eine Anweisung wie

```
y + z;
```

ist trotz (höchstwahrscheinlich) fehlender Wirkung erlaubt. Während der Programmausführung wird die Summe von `y` und `z` berechnet und dann weggeworfen. Neuere Compiler optimieren solche nutzlosen Berechnungen weg. Es wird jedoch nicht garantiert, dass diese Aussage immer weggelassen werden kann. Wenn `y` oder `z` ein Objekt eines Nutzertyps ist, dann ist die Addition ebenfalls nutzerdefiniert und kann `y` oder `z` oder etwas anderes ändern. Dies ist offensichtlich ein schlechter Programmierstil (versteckter Seiteneffekt), aber in C++ erlaubt.

Ein einzelnes Semikolon ist eine leere Anweisung, und wir können so viele Semikolons nach einem Ausdruck setzen, wie wir wollen. Einige Anweisungen enden nicht mit einem Semikolon, z.B. Funktionsdefinitionen. Wenn ein Semikolon an eine solche Anweisung angehängt wird, handelt es sich nicht um einen Fehler, sondern um eine zusätzliche leere Anweisung. Nichtsdestotrotz geben einige Compiler eine Warnung im pedantischen Modus aus. Jede von geschweiften Klammern umgebene Folge von Anweisungen ist auch eine Anweisung. Wir bezeichnen dies als *"Anweisungsblock"* (engl. compound statement).

Die Variablen- und Konstantendeklarationen, die wir zuvor gesehen haben, sind ebenfalls Anweisungen. Als Anfangswert einer Variablen oder Konstanten können wir jeden beliebigen Ausdruck verwenden (außer einem anderen Zuweisungs- oder Komma-Operator). Weitere Anweisungen sind Funktions- und Klassendefinitionen sowie Steueranweisungen, die wir in den nächsten Abschnitten vorstellen werden.

Mit Ausnahme des Bedingungsoperators wird der Programmablauf durch Anweisungen gesteuert. Hier wird zwischen Verzweigungen und Schleifen unterschieden.

1.4.3 Verzweigung

In diesem Abschnitt stellen wir die verschiedenen Features vor, die es uns ermöglichen, einen Verzweigung in der Programmausführung festzulegen.

1.4.3.1 if-Anweisung

Dies ist die einfachste Form der Steuerung und ihre Bedeutung ist intuitiv ersichtlich, z.B. bei:

```
if (weight > 100.0)
    cout ≪ "Das ist recht schwer.\n";
else
    cout ≪ "Das kann ich tragen.\n";
```

Oft wird der else-Zweig nicht benötigt und kann weggelassen werden. Angenommen, wir haben einen Wert in Variable x und berechnen etwas mit seinem Betrag:

```
if (x < 0.0)
    x= -x;
// Jetzt gilt: x >= 0.0 (Nachbedingung)
```

Die Zweige der if-Anweisung sind Geltungsbereiche, was die folgenden Anweisungen fehlerhaft macht:

```
if (x < 0.0)
    int absx= -x;
else
    int absx= x;
cout ≪ "|x| ist " ≪ absx ≪ "\n"; // Fehler: absx nicht mehr im Scope
```

Oben haben wir zwei neue Variablen eingeführt, beide mit dem Namen absx. Sie stehen nicht im Konflikt, weil sie sich in verschiedenen Bereichen befinden. Keiner von beiden existiert nach der if-Anweisung, so dass der Zugriff auf absx in der letzten Zeile ein Fehler ist. Tatsächlich können in einem Zweig deklarierte Variablen nur innerhalb dieses Zweigs verwendet werden.

Jeder Zweig von if besteht aus einer einzigen Anweisung. Um mehrere Operationen durchzuführen, können wir Klammern verwenden, wie im folgenden Beispiel, das die Methode von Cardano realisiert:

```
double D= q*q/4.0 + p*p*p/27.0;
if (D > 0.0) {
    double z1= ...;
    complex<double> z2= ..., z3= ...;
    ...
} else if (D == 0.0) {
    double z1= ..., z2= ..., z3= ...;
    ...
} else {                    // D < 0.0
    complex<double> z1= ..., z2= ..., z3= ...;
    ...
}
```

Am Anfang ist es hilfreich, die Klammern immer zu schreiben. Viele Style-Guides erzwingen auch geschweifte Klammern bei Einzelanweisungen, während der Autor diese ohne Klammern bevorzugt. Unabhängig davon ist es sehr ratsam, die Zweige zur besseren Lesbarkeit einzurücken.

if-Anweisungen können verschachtelt werden, wobei jedes else mit dem letzten offenen if assoziiert ist. Wenn Sie an Beispielen interessiert sind, werfen Sie einen Blick auf Abschnitt A.2.2. Außerdem möchten wir Ihnen noch den folgenden Tipp geben:

Obwohl Leerzeichen die Kompilierung in C++ nicht beeinflussen, sollte die Einrückung die Struktur des Programms wiedergeben. Editoren für C++ mit automatischer Einrückung sind eine große Hilfe bei der strukturierten Programmierung. Immer dann, wenn eine Zeile innerhalb eines sprachkundigen Tools nicht wie erwartet eingerückt wird, ist etwas höchstwahrscheinlich nicht wie vorgesehen geschachtelt.

C++17 Die `if`-Anweisung wurde um die Möglichkeit erweitert, eine Variable zu initialisieren, deren Gültigkeit auf die `if`-Anweisung beschränkt ist. Dies hilft, die Lebensdauer von Variablen zu kontrollieren. Wenn wir beispielsweise einen neuen Wert in eine `map` einfügen, erhalten wir einen Verweis auf den neuen Eintrag und ein `bool`, ob die Einfügung erfolgreich war:

```
map<string, double> constants= {{"e", 2.7}, {"pi", 3.14}};
if (auto res= constants.insert({"h", 6.6e-34}); res.second)
    cout << res.first->first << " eingefügt, mit Abbildung auf "
         << res.first->second << endl;
else
    cout << "Eintrag für " << res.first->first << " existiert bereits.\n";
```

Wir hätten `res` auch vor der `if`-Anweisung deklarieren können und es würde dann bis zum Ende des umgebenden Blocks existieren, es sei denn, wir setzen zusätzliche Klammern um die Variablendeklaration und die `if`-Anweisung.

1.4.3.2 Bedingungsausdruck

Obwohl es in diesem Abschnitt um Anweisungen geht, möchten wir hier auch über den Bedingungsausdruck reden, da er der `if`-Anweisung sehr ähnlich ist. Das Ergebnis von

```
bedingung ? ergebnis_fuer_wahr : ergebnis_fuer_falsch
```

ist der zweite Teilausdruck (d.h. `ergebnis_fuer_wahr`), wenn Bedingung zu wahr evaluiert wird und sonst `ergebnis_fuer_falsch`. Zum Beispiel:

```
min= x <= y ? x : y;
```

entspricht der folgenden `if`-Anweisung:

```
if (x <= y)
    min= x;
else
    min= y;
```

Für einen Anfänger könnte die zweite Version besser lesbar sein, während erfahrene Programmierer oft die erste Form wegen ihrer Kürze bevorzugen. `?:` ist ein Ausdruck und kann daher zur Initialisierung von Variablen verwendet werden:

```
int x= f(a),
    y= x < 0 ? -x : 2 * x;
```

Der Aufruf von Funktionen mit mehreren ausgewählten Argumenten ist mit dem Operator einfach:

```
f(a, (x < 0 ? b : c), (y < 0 ? d : e));
```

aber ziemlich umständlich mit einer if-Anweisung. Wenn Sie uns nicht glauben, versuchen Sie es gerne. In den meisten Fällen ist es nicht so wichtig, ob ein `if` oder `?:` verwendet wird. Nutzen Sie also das, was Ihnen am besten gefällt.

Anekdote: Ein Beispiel, bei dem die Wahl zwischen `if` und `?:` einen Unterschied ausmacht, ist `replace_copy` aus der Standard-Template-Library (STL), Abschnitt 4.1. Früher wurde es mit dem Bedingungsoperator implementiert, während `if` allgemeingültiger ist. Dieser "Fehler" wurde erst durch eine automatische Analyse in Jeremy Sieks Doktorarbeit entdeckt [44] und dass er 10 Jahre unentdeckt blieb, zeigt, dass es sich wohl eher um eine akademische Subtilität als um ein reales Problem handelt.

1.4.3.3 `switch`-Anweisung

Ein `switch` ist wie eine besondere Art von mehreren `if`. Es liefert eine kurze Notation, wenn unterschiedliche Berechnungen für verschiedene Werte eines ganzzahligen Ausdrucks durchgeführt werden sollen:

```
switch(op_code) {
  case 0: z= x + y; break;
  case 1: z= x - y; cout << "berechne diff\n"; break;
  case 2:
  case 3: z= x * y; break;
  default: z= x / y;
}
```

Ein für manchen etwas überraschendes Verhalten ist, dass der Code der folgenden Fälle ebenfalls ausgeführt wird, es sei denn, wir verlassen die Ausführung mit `break`. Daher werden in unserem Beispiel für die Fälle 2 und 3 die gleichen Operationen durchgeführt. Eine Compiler-Warnung für (nicht leere) Fälle ohne `break` wird mit `-Wimplicit-fallthrough` in `g++` and `clang++` erzeugt.

Um solche Warnungen zu vermeiden und den Mitentwicklern mitzuteilen, dass die Fortsetzung beabsichtigt ist, wurde das Attribut `[[fallthrough]]` eingeführt: C++17

```
switch(op_code) {
  case 0: z= x + y; break;
  case 1: z= x - y; cout << "compute diff\n"; break;
  case 2: x= y; [[fallthrough]];
  case 3: z= x * y; break;
  default: z= x / y;
}
```

Außerdem wurde wie für `if` die Möglichkeit hinzugefügt, in der `switch`-Anweisung eine Variable zu initialisieren. Eine fortgeschrittene Anwendung von `switch` finden Sie in Anhang A.2.3. C++17

1.4.4 Schleifen

1.4.4.1 `while`- und `do-while`-Schleifen

Wie der Name schon sagt, wird eine `while`-Schleife wiederholt, solange die gegebene Bedingung erfüllt ist. Implementieren wir als Beispiel die Collatz-Folge, die definiert ist durch:

Input : x_0
while $x_i \neq 1$ **do**
$$x_i = \begin{cases} 3x_{i-1}+1 & \text{if } x_{i-1} \text{ is odd} \\ x_{i-1}/2 & \text{if } x_{i-1} \text{ is even} \end{cases}$$
end

Algorithmus 2 : Collatz-Reihe

Diese Folge lässt sich (bei Vernachlässigung von Überläufen) einfach mit einer `while`-Schleife realisieren:

```
int x= 19;
while (x != 1) {
    cout << x << '\n';
    if (x % 2 == 1)      // ungerade
        x= 3 * x + 1;
    else                 // gerade
        x= x / 2;
}
```

Wie die `if`-Anweisung kann die Schleife ohne geschweifte Klammern geschrieben werden, wenn es nur eine Anweisung gibt.

Im Gegensatz dazu, prüft eine `do-while`-Schleife die Bedingung für die Fortsetzung erst am Ende:

```
double eps= 0.001;
do {
    cout << "eps= " << eps << '\n';
    eps/= 2.0;
} while (eps > 0.0001);
```

Die Schleife wird mindestens einmal ausgeführt.

1.4.4.2 `for`-Schleife

Die häufigste Schleife ist die `for`-Schleife. Als einfaches Beispiel addieren wir zwei Vektoren[6] und drucken das Ergebnis anschließend aus:

```
double v[3], w[]= {2., 4., 6.}, x[]= {6., 5., 4};
for (int i= 0; i < 3; ++i)
    v[i]= w[i] + x[i];

for (int i= 0; i < 3; ++i)
    cout << "v[" << i << "]= " << v[i] << '\n';
```

Der Schleifenkopf besteht aus drei Komponenten:

- der Initialisierung;
- einem Fortsetzungskriterium und
- einer Fortschrittsoperation.

[6] Später werden wir echte Vektorklassen einführen. Im Moment verwenden wir einfache Arrays.