



```
auto [x, y, z] = g(a, b);  
  
(0 + ... + a)  
  
sort(c, [](auto x, auto y) { return x > y; });  
  
auto d5_f = derive<5>(f);  
  
[&s](auto x) { s += x * x; }
```



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



Alle in diesem Buch enthaltenen Informationen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt geprüft und getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autor(en, Herausgeber) und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Weise aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso wenig übernehmen Autor(en, Herausgeber) und Verlag die Gewähr dafür, dass die beschriebenen Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

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.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren) – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2019 Carl Hanser Verlag München

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 geschriebenen 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 geduldgigen 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 architekturenspezifische 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	1
1.1	Unser erstes Programm	1
1.2	Variablen	3
1.2.1	Fundamentale Typen	4
1.2.2	Characters und Strings	5
1.2.3	Variablen deklarieren	6
1.2.4	Konstanten	6
1.2.5	Literale	7
1.2.6	Werterhaltende Initialisierung	9
1.2.7	Gültigkeitsbereiche	10
1.3	Operatoren	12
1.3.1	Arithmetische Operatoren	13
1.3.2	Boolesche Operatoren	15
1.3.3	Bitweise Operatoren	16
1.3.4	Zuweisung.....	17
1.3.5	Programmablauf	18
1.3.6	Speicherverwaltung.....	19
1.3.7	Zugriffsoperatoren	19
1.3.8	Typbehandlung.....	19
1.3.9	Fehlerbehandlung.....	20
1.3.10	Überladung	20
1.3.11	Operatorprioritäten	20
1.3.12	Vermeiden Sie Seiteneffekte!.....	21
1.4	Ausdrücke und Anweisungen	23
1.4.1	Ausdrücke	23
1.4.2	Anweisungen	24
1.4.3	Verzweigung	24
1.4.4	Schleifen	27
1.4.5	<code>goto</code>	30
1.5	Funktionen	31
1.5.1	Argumente	31

1.5.2	Rückgabe der Ergebnisse.....	33
1.5.3	Inlining	34
1.5.4	Überladen	34
1.5.5	Die <code>main</code> -Funktion	36
1.6	Fehlerbehandlung.....	37
1.6.1	Zusicherungen.....	37
1.6.2	Ausnahmen	39
1.6.3	Statische Zusicherungen	43
1.7	I/O	44
1.7.1	Standard-Ausgabe	44
1.7.2	Standard-Eingabe	45
1.7.3	Ein-/Ausgabe mit Dateien	45
1.7.4	Generisches Stream-Konzept.....	46
1.7.5	Formatierung	47
1.7.6	I/O-Fehler behandeln	48
1.7.7	File-System.....	51
1.8	Arrays, Zeiger und Referenzen	52
1.8.1	Arrays	52
1.8.2	Zeiger	54
1.8.3	Intelligente Zeiger	57
1.8.4	Referenzen	60
1.8.5	Vergleich zwischen Zeigern und Referenzen	60
1.8.6	Nicht auf abgelaufene Daten verweisen!	61
1.8.7	Containers for Arrays	62
1.9	Strukturierung von Software-Projekten	64
1.9.1	Kommentare	64
1.9.2	Präprozessor-Direktiven	66
1.10	Aufgaben.....	70
1.10.1	Verengung	70
1.10.2	Literale.....	70
1.10.3	Operatoren	70
1.10.4	Verzweigung	70
1.10.5	Schleifen.....	70
1.10.6	I/O	71
1.10.7	Arrays und Zeiger.....	71
1.10.8	Funktionen	71

2	Klassen	72
2.1	Universell programmieren, nicht detailversessen	72
2.2	Member	74
2.2.1	Mitgliedervariablen	75
2.2.2	Zugriffsrechte	75
2.2.3	Zugriffsoperatoren	78
2.2.4	Der <code>static</code> -Deklarator für Klassen	78
2.2.5	Member-Funktionen	79
2.3	Konstruktoren und Zuweisungen	80
2.3.1	Konstruktoren	80
2.3.2	Zuweisungen	90
2.3.3	Initialisierungslisten	91
2.3.4	Einheitliche Initialisierung	93
2.3.5	Move-Semantik	95
2.3.6	Objekte aus Literalen konstruieren	101
2.4	Destruktoren	103
2.4.1	Implementierungsregeln	104
2.4.2	Richtiger Umgang mit Ressourcen	104
2.5	Zusammenfassung der Methodengenerierung	110
2.6	Zugriff auf Mitgliedervariablen	111
2.6.1	Zugriffsfunktionen	111
2.6.2	Index-Operator	112
2.6.3	Konstante Mitgliederfunktionen	113
2.6.4	Referenz-qualifizierte Mitglieder	115
2.7	Design von Operatorüberladung	116
2.7.1	Seien Sie konsistent!	116
2.7.2	Die Priorität respektieren	117
2.7.3	Methoden oder freie Funktionen	118
2.8	Aufgaben	121
2.8.1	Polynomial	121
2.8.2	Rational	121
2.8.3	Move-Zuweisung	122
2.8.4	Initialisierungsliste	122
2.8.5	Ressourcenrettung	122

3	Generische Programmierung	123
3.1	Funktions-Templates	123
3.1.1	Parametertyp-Deduktion	124
3.1.2	Mit Fehlern in Templates klarkommen	128
3.1.3	Gemischte Typen	129
3.1.4	Einheitliche Initialisierung	130
3.1.5	Automatischer Rückgabotyp	130
3.2	Namensräume und Funktionssuche	131
3.2.1	Namensräume	131
3.2.2	Argumentabhängiges Nachschlagen	134
3.2.3	Namensraum-Qualifizierung oder ADL	138
3.3	Klassen-Templates	140
3.3.1	Ein Container-Beispiel	140
3.3.2	Einheitliche Klassen- und Funktionsschnittstellen entwerfen	142
3.4	Typ-Deduktion und -Definition	148
3.4.1	Automatische Variablentypen	148
3.4.2	Typ eines Ausdrucks	148
3.4.3	<code>decltype(auto)</code>	149
3.4.4	Deduzierte Klassen-Template-Parameter	151
3.4.5	Mehrere Typen deduzieren	152
3.4.6	Typen definieren	153
3.5	Etwas Theorie zu Templates: Konzepte	155
3.6	Template-Spezialisierung	156
3.6.1	Spezialisierung einer Klasse für einen Typ	156
3.6.2	Funktionen spezialisieren und Überladen	158
3.6.3	Partielle Spezialisierung von Klassen	160
3.6.4	Partiell spezialisierte Funktionen	161
3.6.5	Strukturierte Bindung mit Nutzertypen	163
3.7	Nicht-Typ-Parameter für Templates	166
3.7.1	Container fester Größe	166
3.7.2	Nicht-Typ-Parameter deduzieren	169
3.8	Funktoren	169
3.8.1	Funktionsartige Parameter	171
3.8.2	Funktoren zusammensetzen	172
3.8.3	Rekursion	174
3.8.4	Generische Reduktion	177
3.9	Lambdas	178
3.9.1	Objekte erfassen	179

3.9.2	Generische Lambdas	183
3.10	Variablen-Templates	183
3.11	Variadische Templates	185
3.11.1	Rekursive Funktionen	185
3.11.2	Direkte Expansion.....	187
3.11.3	Indexsequenzen	188
3.11.4	Faltung.....	190
3.11.5	Typgeneratoren.....	191
3.11.6	Wachsende Tests	191
3.12	Übungen	193
3.12.1	String-Darstellung	193
3.12.2	String-Darstellung von Tupeln	193
3.12.3	Generischer Stack	194
3.12.4	Rationale Zahlen mit Typparameter.....	194
3.12.5	Iterator eines Vektors	194
3.12.6	Ungerader Iterator	194
3.12.7	Bereich von ungeraden Zahlen.....	195
3.12.8	Stack von <code>bool</code>	195
3.12.9	Stack mit nutzerdefinierter Größe	195
3.12.10	Deduktion von Nicht-Typ-Template-Argumenten.....	195
3.12.11	Trapez-Regel.....	196
3.12.12	Partielle Spezialisierung mit einer statischen Funktion.....	196
3.12.13	Funktor.....	196
3.12.14	Lambda	196
3.12.15	Implementieren Sie <code>make_unique</code>	197
4	Bibliotheken	198
4.1	Standard-Template-Library	199
4.1.1	Einführendes Beispiel	199
4.1.2	Iteratoren	200
4.1.3	Container	205
4.1.4	Algorithmen.....	214
4.1.5	Jenseits von Iteratoren	219
4.1.6	Parallele Berechnung	221
4.2	Numerik.....	222
4.2.1	Komplexe Zahlen.....	222
4.2.2	Zufallszahlengeneratoren.....	225
4.2.3	Mathematische Spezialfunktionen	234

4.3	Meta-Programmierung	235
4.3.1	Wertgrenzen	235
4.3.2	Typeeigenschaften.....	237
4.4	Utilities.....	239
4.4.1	<code>optional</code>	239
4.4.2	Tupel	240
4.4.3	<code>variant</code>	243
4.4.4	<code>any</code>	245
4.4.5	<code>string_view</code>	246
4.4.6	<code>function</code>	247
4.4.7	Referenz-Wrapper	250
4.5	Die Zeit ist gekommen	252
4.6	Parallelität	254
4.6.1	Terminologie.....	254
4.6.2	Überblick	255
4.6.3	Threads	255
4.6.4	Rückmeldung an den Aufrufer	257
4.6.5	Asynchrone Aufrufe	258
4.6.6	Asynchroner Gleichungslöser	260
4.6.7	Variadische Mutex-Sperre.....	264
4.7	Wissenschaftliche Bibliotheken jenseits des Standards.....	266
4.7.1	Andere Arithmetiken	266
4.7.2	Intervallarithmetik	267
4.7.3	Lineare Algebra	267
4.7.4	Gewöhnliche Differentialgleichungen	268
4.7.5	Partielle Differentialgleichungen.....	268
4.7.6	Graphenalgorithmen	269
4.8	Übungen	269
4.8.1	Sortierung nach Betrag	269
4.8.2	Suche mit einem Lambda als Prädikat	269
4.8.3	STL-Container	270
4.8.4	Komplexe Zahlen.....	270
4.8.5	Parallele Vektoraddition	271
4.8.6	Refaktorisierung der parallelen Addition	271

5	Meta-Programmierung	273
5.1	Lassen Sie den Compiler rechnen	273
5.1.1	Kompilierzeitfunktionen	273
5.1.2	Erweiterte Kompilierzeitfunktionen	275
5.1.3	Primzahlen	277
5.1.4	Wie konstant sind unsere Konstanten?	279
5.1.5	Kompilierzeit-Lambdas	280
5.2	Typinformationen	281
5.2.1	Typabhängige Funktionsergebnisse	281
5.2.2	Bedingte Ausnahmebehandlung	285
5.2.3	Ein Beispiel für eine <code>const</code> -korrekte View	286
5.2.4	Standard-Typmerkmale	293
5.2.5	Domän-spezifische Type-Traits	293
5.2.6	Typeigenschaften mit Überladung	295
5.2.7	<code>enable_if</code>	297
5.2.8	Variadische Templates überarbeitet	301
5.3	Expression-Templates	304
5.3.1	Einfache Implementierung eines Additionsoperators	304
5.3.2	Eine Klasse für Expression-Templates	308
5.3.3	Generische Expression-Templates	310
5.4	Compiler-Optimierung mit Meta-Tuning	312
5.4.1	Klassisches Abrollen mit fester Größe	313
5.4.2	Geschachteltes Abrollen	317
5.4.3	Aufwärmung zum dynamischen Abrollen	323
5.4.4	Abrollen von Vektorausdrücken	324
5.4.5	Tuning von Expression-Templates	326
5.4.6	Tuning von Reduktionen	329
5.4.7	Tuning geschachtelter Schleifen	336
5.4.8	Resümee des Tunings	341
5.5	Turing-Vollständigkeit	343
5.6	Übungen	346
5.6.1	Type-Traits	346
5.6.2	Fibonacci-Sequenz	346
5.6.3	Meta-Programm für den größten gemeinsamen Divisor	346
5.6.4	Rationale Zahlen mit gemischten Typen	347
5.6.5	Vektor-Expression-Template	347
5.6.6	Meta-Liste	348

6	Objektorientierte Programmierung	349
6.1	Grundprinzipien	349
6.1.1	Basis- und abgeleitete Klassen	350
6.1.2	Konstruktorerben	353
6.1.3	Virtuelle Funktionen	354
6.1.4	Funktoren über Vererbung	361
6.1.5	Abgeleitete Klassen für Ausnahmen	362
6.2	Redundanz entfernen	363
6.3	Mehrfachvererbung	365
6.3.1	Mehrere Eltern	365
6.3.2	Gemeinsame Großeltern	366
6.4	Dynamische Auswahl von Subtypen	371
6.5	Konvertierung	373
6.5.1	Umwandlungen zwischen abgeleiteten Klassen	374
6.5.2	<code>const_cast</code>	378
6.5.3	Umdeutung	379
6.5.4	Umwandlung im Funktionsstil	379
6.5.5	Implizite Umwandlungen	381
6.6	CRTP	382
6.6.1	Ein einfaches Beispiel	382
6.6.2	Ein wiederverwendbarer Indexoperator	384
6.7	Übungen	386
6.7.1	Nicht-redundante Raute	386
6.7.2	Vektorklasse mit Vererbung	386
6.7.3	Ausnahmen in Vektor refaktorisieren	386
6.7.4	Test auf geworfene Ausnahme	387
6.7.5	Klonfunktion	387
7	Wissenschaftliche Projekte	388
7.1	Implementierung von ODE-Lösern	388
7.1.1	Gewöhnliche Differentialgleichungen	388
7.1.2	Runge-Kutta-Algorithmen	391
7.1.3	Generische Implementierung	392
7.1.4	Ausblick	399
7.2	Projekte erstellen	399
7.2.1	Build-Prozess	400
7.2.2	Build-Tools	404
7.2.3	Separates Kompilieren	408
7.3	Einige abschließende Worte	414

A Weitschweifendes	416
A.1 Mehr über gute und schlechte Software.....	416
A.2 Grundlagen im Detail	422
A.2.1 Statische Variablen	422
A.2.2 Mehr über <code>if</code>	422
A.2.3 Duff's Device	424
A.2.4 Programmaufrufe	424
A.2.5 Zusicherung oder Ausnahme?	425
A.2.6 Binäre I/O	426
A.2.7 I/O im Stile von C	427
A.2.8 Garbage-Collection	428
A.2.9 Ärger mit Makros	429
A.3 Praxisbeispiel: Matrix-Invertierung	430
A.4 Klassendetails	440
A.4.1 Zeiger auf Mitglieder	440
A.4.2 Weitere Initialisierungsbeispiele	440
A.4.3 Zugriff auf mehrdimensionale Datenstrukturen.....	441
A.5 Methodengenerierung.....	444
A.5.1 Automatische Generierung	445
A.5.2 Steuerung der Generierung.....	447
A.5.3 Generierungsregeln	448
A.5.4 Fallstricke und Designrichtlinien	452
A.6 Template-Details	456
A.6.1 Einheitliche Initialisierung.....	456
A.6.2 Welche Funktion wird aufgerufen?	456
A.6.3 Spezialisierung auf spezifische Hardware	459
A.6.4 Variadisches binäres I/O	460
A.7 Mehr über Bibliotheken	461
A.7.1 <code>std::vector</code> in C++03 verwenden	461
A.7.2 <code>variant</code> mal nerdisch	462
A.8 Dynamische Auswahl im alten Stil	462
A.9 Mehr über Meta-Programmierung.....	463
A.9.1 Das erste Meta-Programm in der Geschichte	463
A.9.2 Meta-Funktionen.....	465
A.9.3 Rückwärtskompatible statische Zusicherung	466
A.9.4 Anonyme Typparameter	467
A.9.5 Benchmark-Quellen für dynamisches Abrollen	471
A.9.6 Benchmark für Matrixprodukt	472

B	Werkzeuge	473
B.1	g++	473
B.2	Debugging	474
B.2.1	Textbasierte Debugger	474
B.2.2	Debugging mit graphischen Interface: DDD	476
B.3	Speicheranalyse	478
B.4	gnuplot	479
B.5	Unix, Linux und Mac OS	480
C	Sprachdefinitionen	482
C.1	Wertkategorien	482
C.2	Konvertierungsregeln	485
C.2.1	Aufwertung	486
C.2.2	Andere Konvertierungen	486
C.2.3	Arithmetische Konvertierungen	487
C.2.4	Verengung	488
	Literatur	489
	Abbildungsverzeichnis	492
	Tabellenverzeichnis	493
	Index	494

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 aufrufen, 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-System¹ 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 Warte-phase 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
<code>char</code>	Buchstabe oder sehr kleine Zahl
<code>short</code>	recht kleine ganze Zahl
<code>int</code>	reguläre ganze Zahl
<code>long</code>	große ganze Zahl
<code>long long</code>	sehr große ganze Zahl
<code>unsigned</code>	vorzeichenlose Versionen der vorangegangenen
<code>signed</code>	vorzeichenbehaftete Versionen der vorangegangenen
<code>float</code>	Gleitkommazahl mit einfacher Genauigkeit
<code>double</code>	Gleitkommazahl mit doppelter Genauigkeit
<code>long double</code>	Gleitkommazahl mit mehr als doppelter Genauigkeit
<code>bool</code>	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";
}
```