BJARNE STROUSTRUP

Eine Tour durch C++

Der praktische Leitfaden für modernes C++

mitp

Die neuesten Sprachfeatures im Überblick
Verfasst vom Entwickler von C++
Übersetzung der 3. Auflage

Bjarne Stroustrup

Eine Tour durch C++

Der praktische Leitfaden für modernes C++

Übersetzung aus dem Englischen von
Kathrin Lichtenberg

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.

Bei der Herstellung des Werkes haben wir uns zukunftsbewusst für umweltverträgliche und wiederverwertbare Materialien entschieden.

Der Inhalt ist auf elementar chlorfreiem Papier gedruckt.

ISBN 978-3-7475-0626-4

1. Auflage 2023

www.mitp.de

E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

Lektorat: Janina Bahlmann
Fachkorrektorat: Philipp Hasper, Conny Lichtenberg
Sprachkorrektorat: Petra Heubach-Erdmann
Covergestaltung: Christian Kalkert
Coverbild: Marco Pregnolato (Unsplash.com: @marco_pregnolato)
Satz: III-satz, Kiel, www.drei-satz.de

Inhaltsverzeichnis

Einleitung

Was auch immer du lehren wirst, fasse dich kurz.
– Horaz, Ars poetica 335

C++ fühlt sich an wie eine neue Sprache. Das heißt, man kann Ideen heute deutlicher, leichter und direkter ausdrücken als in C++98 oder C++11. Außerdem werden die daraus entstehenden Programme besser vom Compiler überprüft und laufen schneller.

Dieses Buch bietet Ihnen einen Überblick über das C++, das durch C++20, den aktuellen ISO-C++-Standard, definiert und durch die wichtigsten Anbieter von C++ implementiert wird. Darüber hinaus werden eine Reihe von Bibliothekskomponenten erwähnt, die momentan schon in Gebrauch sind, aber erst mit C++23 in den Standard aufgenommen werden sollen.

Wie andere moderne Sprachen ist C++ umfangreich und es sind viele Bibliotheken erforderlich, um es effektiv benutzen zu können. Dieses recht schmale Buch soll erfahrenen Programmierern eine Vorstellung davon vermitteln, was modernes C++ ausmacht. Es behandelt die wichtigsten Eigenschaften der Sprache sowie die wichtigsten Komponenten der Standardbibliothek. Es ist möglich, das Buch in ein oder zwei Tagen durchzulesen, aber natürlich braucht man mehr als zwei Tage, um zu lernen, gutes C++ zu schreiben. Das Ziel ist hier aber nicht, C++ zu beherrschen. Stattdessen erhalten Sie einen Überblick, zentrale Beispiele und eine gute Ausgangsbasis.

Ich gehe davon aus, dass Sie bereits programmiert haben. Falls nicht, sollten Sie zuerst ein Lehrbuch wie *Programming: Principles and Practice Using C++ (Second edition)* [Stroustrup, 2014] lesen, bevor Sie hier weitermachen[1]. Selbst wenn Sie programmiert haben, könnten die von Ihnen benutzte Sprache oder die von Ihnen geschriebenen Anwendungen sich grundlegend von dem Stil des C++ unterscheiden, der hier vorgestellt wird.

Stellen Sie sich eine Besichtigungstour in einer Stadt wie Kopenhagen oder New York vor. In nur wenigen Stunden erhaschen Sie einen kurzen Blick auf die wichtigsten

1 Anm. zur Übersetzung: Auch im deutschsprachigen Raum sind geeignete Lehrbücher erschienen, beispielsweise *C++ Schnelleinstieg* von (Hasper, 2021) oder *C++ lernen und professionell anwenden* (Prinz/Kirch, 2022).

Sehenswürdigkeiten, hören ein paar Anekdoten und bekommen Vorschläge, was Sie als Nächstes tun könnten. Sie kennen die Stadt nach einer solchen Rundfahrt *nicht*. Sie verstehen *nicht* alles, was Sie gesehen und gehört haben; manche der Geschichten klingen vermutlich seltsam oder sogar unglaubwürdig. Sie kennen auch *nicht* die offiziellen und inoffiziellen Regeln, die das Leben in der Stadt bestimmen. Um eine Stadt wirklich kennenzulernen, müssen Sie darin leben, am besten für viele Jahre. Mit ein bisschen Glück haben Sie allerdings einen Überblick gewonnen, ein Gefühl dafür, was so besonders an der Stadt ist, und können sich vielleicht vorstellen, was für Sie interessant sein könnte. Nach der Tour kann die eigentliche Entdeckungsreise beginnen.

Diese Tour stellt die wichtigsten C++-Spracheigenschaften vor, die Programmierparadigmen unterstützen, wie die objektorientierte und die generische Programmierung. Sie versucht nicht, einen detaillierten, alle Funktionen und Eigenschaften einschließenden Blick auf die Sprache zu liefern – dieses Buch soll kein Referenzhandbuch sein. In bester Lehrbuchtradition versuche ich, ein Feature zu erklären, bevor ich es benutze, aber das ist nicht immer möglich und nicht jeder liest einen Text streng sequenziell. Ich erwarte von meinen Leserinnen und Lesern eine gewisse technische Reife. Sie sind eingeladen, die Querverweise und den Index zu benutzen.

Auch die Standardbibliotheken werden auf dieser Tour nicht allumfassend, sondern nur beispielhaft vorgestellt. Suchen Sie bei Bedarf selbst nach zusätzlichen und unterstützenden Materialien. Das C++-Ökosystem bietet viel mehr als nur die Möglichkeiten, die der ISO-Standard mitbringt (z. B. Bibliotheken, Build-Systeme, Analysewerkzeuge und Entwicklungsumgebungen). Es gibt im Web eine Unmenge an Material (von durchaus unterschiedlicher Qualität). Die Tutorial- und Überblicksvideos von Konferenzen wie CppCon und Meeting C++ werden viele Leserinnen und Leser sicher überaus nützlich finden. Für die technischen Details der Sprache und Bibliothek, die vom ISO-C++-Standard angeboten werden, empfehle ich [Cppreference]. Wenn ich zum Beispiel eine Funktion oder Klasse der Standardbibliothek erwähne, kann deren Definition leicht nachgeschlagen werden. Und in der Dokumentation lassen sich dann auch viele weitere, damit verwandte Möglichkeiten finden.

Diese Tour präsentiert C++ als geschlossenes Ganzes. Entsprechend gebe ich nur selten an, ob Sprachmerkmale zu C, C++98 oder späteren ISO-Standards gehören. Solche Informationen finden Sie in Kapitel 19 (Geschichte und Kompatibilität). Ich konzentriere mich auf die Grundlagen und versuche, mich kurz zu fassen, konnte aber dennoch nicht der Versuchung widerstehen, neue Eigenschaften, wie Module (§3.2.2), Konzepte (§8.2) und Coroutinen (§18.6), ausführlicher zu behandeln. Dass der Schwerpunkt eher auf neueren Entwicklungen liegt, wird auch die Neugier vieler Leserinnen und Leser befriedigen, die bereits ältere Versionen von C++ kennen.

Das Referenzhandbuch oder der Standard einer Sprache hält einfach nur fest, was gemacht werden kann. Programmiererinnen und Programmierer wollen jedoch oft lieber lernen, wie sie die Sprache gut einsetzen können. Diesem Aspekt wird durch die Auswahl der behandelten Themen Genüge getan – zum Teil im Text, vor allem aber in den Abschnitten mit den Ratschlägen. Weitere Hinweise dazu, was gutes, modernes C++ ausmacht, können Sie in den C++ Core Guidelines [Stroustrup, 2015] finden. Die Core Guidelines eignen sich hervorragend, um die in diesem Buch vorgestellten Ideen weiter zu erkunden. Sie werden vermutlich eine bemerkenswerte Ähnlichkeit zwischen der Formulierung und sogar der Nummerierung der Ratschläge in den Core Guidelines und diesem Buch bemerken. Ein Grund dafür ist, dass die erste Auflage von *A Tour of C++* eine wesentliche Quelle für die ersten Core Guidelines war.

Danksagungen

Ein Dank geht an alle, die geholfen haben, die früheren Ausgaben von *A Tour of C++* fertigzustellen und zu korrigieren, vor allem die Studentinnen und Studenten in meinem »Design Using C++«-Kurs an der Columbia University. Ich danke Morgan Stanley, dass sie mir die Zeit gegeben hat, diese dritte Auflage zu verfassen. Danke an Chuck Allison, Guy Davidson, Stephen Dewhurst, Kate Gregory, Danny Kalev, Gor Nishanov und J. C. van Winkel für das Begutachten des Buches und die vielen Verbesserungsvorschläge.

Die Originalausgabe dieses Buches wurde vom Autor mit troff gesetzt, die verwendeten Makros stammten von Brian Kernighan.

Manhattan, New York
Bjarne Stroustrup

Über die Fachkorrektoren der deutschen Ausgabe

Philipp Hasper ist Gründer eines Augmented-Reality-Startups und erfahren in der akademischen und industriellen Entwicklung von KI-Technologien. Er entwickelt mit C++, Java, Python und Typescript und hat bei zahlreichen Open-Source-Projekten mitgewirkt. Von ihm stammt auch das Buch *C++ Schnelleinstieg*, das im mitp-Verlag erschienen ist.

Conny Lichtenberg widmete sich nach seinem Informatikstudium für viele Jahre in seiner eigenen kleinen Firma dem informationstechnischen Allerlei – Systementwurf, Programmierung, Consulting –, bevor er sich neue Herausforderungen suchte und nun Softwareprojekte betreut. Seine Spezialität ist das Werkeln auf der Kommandozeile und er hat den Ehrgeiz, möglichst viele Probleme mit kunstvoll konstruierten regulären Ausdrücken und Pipelines zu lösen.

Kapitel 1

Die Grundlagen

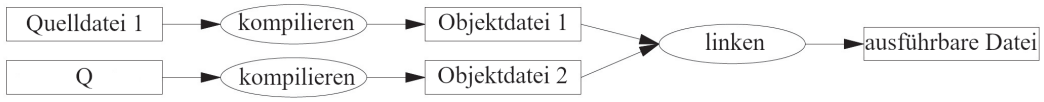
The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II

1.1 Einführung

Dieses Kapitel präsentiert ganz formlos die Notation von C++, das Speicher- und Berechnungsmodell von C++ sowie die grundlegenden Mechanismen, nach denen Code zu einem Programm zusammengefügt wird. Dies sind die Komponenten, die man vor allem in C sieht und die einen Programmierstil bilden, der als *prozedurale Programmierung* bezeichnet wird.

1.2 Programme

C++ ist eine kompilierte Sprache. Damit ein Programm ausgeführt werden kann, muss sein Quelltext durch einen Compiler verarbeitet werden. Dabei werden Objektdateien erzeugt, die dann ein Linker zu einem ausführbaren Programm kombiniert. Ein C++-Programm besteht typischerweise aus vielen Quellcodedateien (meist einfach *Quelldateien* genannt).



Ein ausführbares Programm wird für eine bestimmte Hardware/System-Kombination erzeugt; es kann nicht von z. B. einem Android-Gerät auf einen Windows-PC übertragen werden. Wenn es um die Portabilität von C++-Programmen geht, dann meinen wir üblicherweise die Portabilität des Quellcodes; das heißt, dass der Quellcode erfolgreich auf einer Vielzahl von Systemen kompiliert und ausgeführt werden kann.

Der ISO-C++-Standard definiert zwei Arten von Entitäten:

- *Elemente der Kernsprache*, wie integrierte Typen (z. B. **`char`** und **`int`**) und Schleifen (z. B. **`for`**- und **`while`**-Anweisungen)
- *Komponenten der Standardbibliothek*, wie etwa Container (z. B. **`vector`** und **`map`**) und I/O-Operationen (z. B. **`<<`** und **`getline()`**)

Bei den Komponenten der Standardbibliothek handelt es sich um völlig normalen C++-Code, der von jeder C++-Implementierung bereitgestellt wird. Das heißt, die C++-Standardbibliothek kann selbst in C++ implementiert werden, was auch so ist (mit sehr geringfügigem Einsatz von Maschinencode für Dinge wie **`thread`**-Kontextwechsel). Das impliziert, dass C++ für die anspruchsvollsten Aufgaben im Bereich der Systemprogrammierung ausreichend ausdrucksstark und effizient ist.

C++ gehört zu den statisch typisierten Sprachen. Das heißt, der Typ jeder Entität (wie etwa Objekt, Wert, Name und Ausdruck) muss dem Compiler an der Stelle bekannt sein, an der sie benutzt wird. Der Typ eines Objekts bestimmt die Menge der Operationen, die darauf angewendet werden können, sowie seine Anordnung im Speicher.

1.2.1 Hello, World!

Das kleinstmögliche C++-Programm ist

```
int main(){}        // das kleinstmögliche C++-Programm
```

Es definiert eine Funktion namens **`main()`**, die keine Argumente entgegennimmt und nichts tut.

Geschweifte Klammern, **`{}`**, drücken in C++ eine Gruppierung aus. Hier kennzeichnen sie den Anfang und das Ende des Funktionskörpers. Der doppelte Schrägstrich, **`//`**, startet einen Kommentar, der bis zum Zeilenende reicht. Ein Kommentar ist für die menschlichen Leserinnen und Leser vorgesehen; der Compiler ignoriert Kommentare.

Jedes C++-Programm muss genau eine globale Funktion namens **`main()`** besitzen. Das Programm startet, indem es diese Funktion ausführt. Der Integer-Wert **`int`**, der von **`main()`** zurückgegeben wird, falls er vorhanden ist, ist der Rückgabewert des Programms an »das System«. Wird kein Wert zurückgegeben, erhält das System einen Wert, der einen erfolgreichen Abschluss des Programms signalisiert. Ist der von **`main()`** zurückgegebene Wert ungleich null, bedeutet dies ein Fehlschlagen des Programms. Nicht alle Betriebssysteme und Ausführungsumgebungen machen Gebrauch von diesem Rückgabewert: Linux/Unix-Systeme tun es, Windows-Umgebungen dagegen nur selten.

Üblicherweise erzeugt ein Programm irgendeine Ausgabe. Hier ist ein Programm, das **`Hello, World!`** schreibt:

```
import std;

int main()
{
     std::cout << "Hello, World!\n";
}
```

Die Zeile **`import std;`** weist den Compiler an, die Deklarationen der Standardbibliothek zur Verfügung zu stellen. Ohne diese Deklarationen wäre der Ausdruck

```
std::cout << "Hello, World!\n"
```

sinnlos. Der Operator **`<<`** (»ausgeben«) schreibt sein zweites Argument auf sein erstes. In diesem Fall wird das String-Literal **`"Hello, World!\n"`** auf den Standard-Ausgabe-Stream **`std::cout`** geschrieben. Ein String-Literal ist eine Folge von Zeichen, die von doppelten Anführungszeichen umgeben sind. In einem String-Literal kennzeichnet der Backslash **`\`** gefolgt von einem anderen Zeichen ein einzelnes »Sonderzeichen«. Hier ist **`\n`** das Newline-Zeichen. Es werden also die Zeichen **`Hello, World!`** geschrieben, gefolgt von einem Newline, also dem Steuerzeichen für eine neue Zeile.

`std::` gibt an, dass der Name (Bezeichner) **`cout`** im Namensraum der Standardbibliothek (§3.3) zu finden ist. Ich lasse das **`std::`** normalerweise weg, wenn es um Standardeigenschaften geht. §3.3 zeigt, wie man Namen aus einem Namensraum auch ohne explizite Qualifizierung sichtbar machen kann.

Die Direktive **`import`** ist neu in C++20. Es ist noch nicht im Standard verankert, dass die gesamte Standardbibliothek als Modul **`std`** vorhanden ist. Das wird in §3.2.2 erklärt. Falls Sie Probleme mit **`import std;`** haben, probieren Sie das altmodische und herkömmliche

```
#include <iostream>        // bindet die Deklarationen für die
                           // I/O-Stream-Bibliothek ein

int main()
{
     std::cout << "Hello, World!\n";
}
```

Das wird in §3.2.1 erklärt und hat in allen C++-Implementierungen seit 1998 funktioniert (§19.1.1).

Im Prinzip wird der gesamte ausführbare Code in Funktionen gepackt und direkt oder indirekt aus **`main()`** heraus aufgerufen. Zum Beispiel:

```
import std;            // importiert die Deklarationen für die
                       // Standardbibliothek
using namespace std;    // macht die Namen aus std auch ohne
                        // std:: sichtbar (§3.3)
double square(double x)    // quadriert eine Gleitkommazahl mit doppelter
                           // Genauigkeit
{
     return x*x;
}

void print_square(double x)
{
     cout << "das Quadrat von " << x << " ist " << square(x) << "\n";
}

int main()
{
     print_square(1.234) // Ausgabe: das Quadrat von 1,234 ist 1,52276
}
```

Der »Rückgabetyp« **`void`** zeigt an, dass die Funktion keinen Wert zurückgibt.

1.3 Funktionen

Die wichtigste Möglichkeit, irgendetwas in einem C++-Programm erledigen zu lassen, besteht darin, dafür eine Funktion aufzurufen. Über das Definieren einer Funktion legen Sie fest, wie eine Operation durchgeführt werden soll. Eine Funktion kann nur aufgerufen werden, wenn sie zuvor deklariert wurde.

Eine Funktionsdeklaration legt den Namen der Funktion, den Typ des zurückgelieferten Werts (falls vorhanden) und die Anzahl und Typen der Argumente fest, die in einem Aufruf angegeben werden müssen. Zum Beispiel:

```
Elem* next_elem();       // kein Argument, liefert einen Zeiger auf
                         // Elem (einen Elem*) zurück
void exit(int);          // int-Argument, liefert nichts zurück
double sqrt(double);     // double-Argument, liefert einen double zurück
```

In einer Funktionsdeklaration steht der Rückgabetyp vor dem Namen der Funktion; die Argumenttypen stehen hinter dem Namen und werden in Klammern eingeschlossen.

Die Semantik der Argumentübergabe ist identisch mit der Semantik der Initialisierung (§3.4.1). Das heißt, die Argumenttypen werden geprüft und falls notwendig findet eine implizite Konvertierung der Argumenttypen statt (§1.4). Zum Beispiel:

```
double s2 = sqrt(2);  // Aufruf von sqrt() mit dem Argument double{2}
double s3 = sqrt("three");  // Fehler: sqrt() verlangt ein Argument des
                            // Typs double
```

Man sollte den Wert einer solchen Prüfung und Typkonvertierung zum Compile-Zeitpunkt nicht unterschätzen.

Eine Funktionsdeklaration könnte Argumentnamen enthalten. Dies kann für den Leser eines Programms hilfreich sein, doch der Compiler ignoriert solche Namen einfach, solange die Deklaration nicht auch eine Funktionsdefinition ist. Zum Beispiel:

```
double sqrt(double d);        // gibt die Quadratwurzel von d zurück
double square(double);        // gibt das Quadrat des Arguments zurück
```

Der Typ einer Funktion besteht aus ihrem Rückgabetyp, gefolgt von einer Abfolge ihrer Argumenttypen in runden Klammern. Zum Beispiel:

```
double get(const vector<double>& vec, int index);   // Typ: double(const
                                                     // vector<double>&,int)
```

Eine Funktion kann Member (Mitglied) einer Klasse sein (§2.3, §5.2.1). Bei einer solchen Member-Funktion ist der Name ihrer Klasse ebenfalls Teil des Funktionstyps. Zum Beispiel:

```
char& String::operator[](int index);    // Typ: char& String::(int)
```

Wir wollen, dass unser Code verständlich ist, weil dies den ersten Schritt auf dem Weg zur Wartungsfreundlichkeit bedeutet. Um Verständlichkeit zu erreichen, zerlegt man als Erstes die Berechnungsaufgaben in sinnvolle Einheiten (dargestellt als Funktionen und Klassen) und benennt diese. Solche Funktionen bilden dann das Grundvokabular der rechnerischen Verarbeitung, genau wie die (integrierten und benutzerdefinierten) Typen das Grundvokabular der Daten bilden. Die C++-Standardalgorithmen (z. B. **`find`**, **`sort`** und **`iota`**) sind ein guter Start (Kapitel 13).

Anschließend können Sie Funktionen, die gängige oder spezialisierte Aufgaben repräsentieren, zu größeren Verarbeitungseinheiten zusammensetzen.

Die Anzahl der Fehler in Code korreliert stark mit der Menge und der Komplexität des Codes. Beiden Problemen können Sie begegnen, indem Sie mehr und kürzere Funktionen verwenden. Eine Funktion zu benutzen, die eine bestimmte Aufgabe erledigt, erspart es oft, mitten in irgendwelchem Code ein spezialisiertes Stück Code schreiben zu müssen; wenn Sie daraus eine Funktion bauen, sind Sie gezwungen, die Aktivität zu benennen und ihre Abhängigkeiten zu dokumentieren. Können Sie keinen passenden Namen finden, dann ist es sehr wahrscheinlich, dass Sie ein Designproblem haben.

Falls zwei Funktionen mit demselben Namen, aber unterschiedlichen Argumenttypen definiert sind, wählt der Compiler bei jedem Aufruf die Funktion, die am passendsten erscheint. Zum Beispiel:

```
void print(int);           // nimmt ein Integer-Argument entgegen
void print(double);        // nimmt ein Gleitkomma-Argument entgegen
void print(string);        // nimmt ein String-Argument entgegen

void user()
{
     print(42);            // ruft print(int) auf
     print(9.65);          // ruft print(double) auf
     print("Barcelona");   // ruft print(string) auf
}
```

Falls zwei alternative Funktionen aufgerufen werden könnten, aber keine von beiden besser als die andere ist, dann gilt der Aufruf als mehrdeutig und der Compiler gibt einen Fehler aus. Zum Beispiel:

```
void print(int, double);
void print(double, int);

void user2()
{
     print(0,0);        // Fehler: mehrdeutig
}
```

Das Definieren mehrerer Funktionen mit demselben Namen wird als *Überladen der Funktion* bezeichnet. Es ist ein wesentlicher Bestandteil der generischen Programmierung (§8.2). Wenn eine Funktion überladen wird, dann sollten alle Funktionen mit demselben Namen die gleiche Semantik implementieren. Die **`print()`**-Funktionen sind ein Beispiel dafür; jedes **`print()`** gibt sein Argument aus.

1.4 Typen, Variablen und Arithmetik

Jeder Name und jeder Ausdruck hat einen Typ, der bestimmt, welche Operationen darauf ausgeführt werden dürfen. So legt zum Beispiel die Deklaration

```
int inch;
```

fest, dass **inch** vom Typ **int** ist; das heißt, **inch** ist eine Integer-Variable.

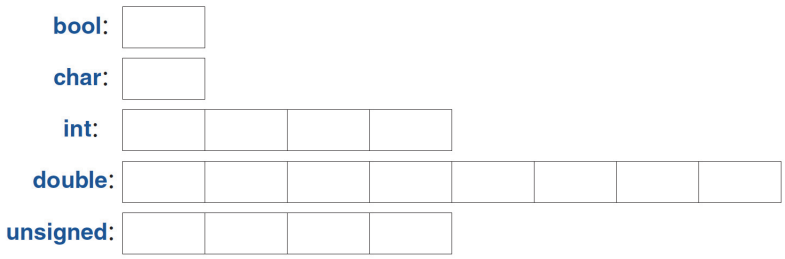
Eine Deklaration ist eine Anweisung, die eine Entität in das Programm einführt und ihren Typ festlegt:

- Ein *Typ* definiert eine Menge an möglichen Werten und eine Menge an Operationen (für ein Objekt).
- Ein *Objekt* ist ein Speicherbereich, der einen Wert eines bestimmten Typs enthält.
- Ein *Wert* ist eine Menge an Bits, die entsprechend einem Typ interpretiert werden.
- Eine *Variable* ist ein benanntes Objekt.

C++ bietet eine ganze Reihe grundlegender Typen, die ich hier aber nicht alle aufführen will. Sie können sie in Referenzquellen finden, etwa in [Cppreference] im Netz. Hier nur einige Beispiele:

```
bool        // Boolean, mögliche Werte sind true und false
char        // Zeichen, zum Beispiel 'a', 'z' und '9'
int         // Integer, zum Beispiel -273, 42 und 1066
double      // Gleitkommazahl doppelter Genauigkeit, zum Beispiel
            // -273.15, 3.14 und 6.626e-34
unsigned    // nichtnegativer Integer, zum Beispiel 0, 1 und 999
            // (wird für bitweise logische Operationen benutzt)
```

Jeder grundlegende Typ besitzt direkt eine Hardware-Entsprechung und hat eine feste Größe, die den Wertebereich festlegt, der darin gespeichert werden kann:



Eine **char**-Variable hat die natürliche Größe eines Zeichens auf einem bestimmten Computer (üblicherweise handelt es sich um ein 8 Bit langes Byte). Die Größen der anderen Typen sind Vielfaches der Größe eines **char**. Die Größe eines Typs ist von der Implementierung abhängig (d. h. kann auf unterschiedlichen Maschinen verschieden ausfallen) und lässt sich durch den **sizeof**-Operator ermitteln. So ist zum Beispiel **sizeof(char)** gleich **1**, während **sizeof(int)** oft **4** beträgt. Wenn Sie einen Typ einer bestimmten Größe haben wollen, benutzen Sie einen Typ-Alias der Standardbibliothek, wie etwa **int32_t** (§17.8).

Zahlen können als Gleitkommazahlen oder als Integer-Werte vorliegen.

- Gleitkomma-Literale sind an einem Dezimalpunkt (z. B. **3.14**) oder einem Exponenten (z. B. **314e-2**) erkennbar.
- Integer-Literale sind standardmäßig dezimal (z. B. **42** bedeutet zweiundvierzig). Das Präfix **0b** kennzeichnet ein binäres (Basis 2) Integer-Literal (z. B. **0b10101010**). Das Präfix **0x** kennzeichnet ein hexadezimales (Basis 16) Integer-Literal (z. B. **0xBAD12CE3**). Das Präfix **0** kennzeichnet ein oktales (Basis 8) Integer-Literal (z. B. **0334**).

Damit lange Literale für uns Menschen besser lesbar sind, können Sie ein einfaches Anführungszeichen (') als Trennzeichen benutzen. So beträgt zum Beispiel π ungefähr **3.14159'26535'89793'23846'26433'83279'50288** oder, falls Sie die hexadezimale Notation bevorzugen, **0x1.921F'B544'42D1'8P+1**.

1.4.1 Rechenoperatoren

Die arithmetischen Operatoren können für geeignete Kombinationen der Grundtypen benutzt werden:

```
x+y       // Plus
+x        // unäres Plus
x-y       // Minus
-x        // unäres Minus
x*y       // Multiplizieren
x/y       // Dividieren
x%y       // Rest (Modulo) für Integer
```

Das gilt auch für Vergleichsoperatoren:

```
x==y       // Gleich
x!=y       // Ungleich
x<y        // Kleiner als
x>y        // Größer als
x<=y       // Kleiner als oder gleich
x>=y       // Größer als oder gleich
```

Es gibt darüber hinaus auch Logikoperatoren:

```
x&y       // Bitweises Und
x|y       // Bitweises Oder
x^y       // Bitweises Exklusiv-Oder
~ x       // Bitweises Komplement
x&&y      // Logisches Und
x||y      // Logisches Oder
! x       // Logisches Nicht (Negation)
```

Ein bitweiser logischer Operator liefert als Ergebnis den Operandentyp, für den die Operation auf jedem Bit durchgeführt wurde. Die Logikoperatoren **&&** und **||** geben je nach den Werten ihrer Operanden einfach **`true`** oder **`false`** zurück.

In Zuweisungen und arithmetischen Operationen führt C++ alle sinnvollen Konvertierungen zwischen den Grundtypen durch, sodass diese frei gemischt werden können:

```
void some_function() // Funktion, die keinen Wert zurückliefert
{
     double d = 2.2; // Initialisiert eine Gleitkommazahl
     int i = 7;      // Initialisiert Integer
     d = d+i;        // Weist d eine Summe zu
     i = d*i;        // Weist i ein Produkt zu; Achtung: das double d*i
                     // wird zu einem int abgeschnitten
}
```

Die in Ausdrücken benutzten Konvertierungen werden als *die üblichen arithmetischen Konvertierungen* bezeichnet und sollen sicherstellen, dass die Ausdrücke mit der höchsten Genauigkeit ihrer Operanden verarbeitet werden. So wird zum Beispiel eine Addition eines **`double`** und eines **`int`** mittels Gleitkomma-Arithmetik mit doppelter Genauigkeit ausgeführt.

Beachten Sie, dass **=** der Zuweisungsoperator ist, **==** dagegen auf Gleichheit prüft.

Zusätzlich zu den herkömmlichen arithmetischen und logischen Operatoren bietet C++ speziellere Operationen zum Modifizieren einer Variablen:

```
x+=y      // x = x+y
++x       // Inkrementiert: x = x+1
x-=y      // x = x-y
-- x      // Dekrementiert: x = x-1
x*=y      // Skaliert: x = x*y
x/=y      // Skaliert: x = x/y
x%=y      // x = x%y
```

Diese Operatoren sind kompakt, bequem und werden sehr häufig verwendet.

Die Auswertung erfolgt von links nach rechts für **`x.y`**, **`x->y`**, **`x(y)`**, **`x[y]`**, **`x<<y`**, **`x>>y`**, **`x&&y`** und **`x||y`**. Zuweisungen (z. B. **`x+=y`**) werden von rechts nach links ausgewertet. Aus historischen Gründen, die mit dem Drang nach Optimierung zusammenhängen, ist die Auswertungsreihenfolge von anderen Ausdrücken (z. B. **`f(x) + g(y)`**) sowie von Funktionsargumenten (z. B. **`h(f(x),g(y))`**) leider nicht festgelegt.

1.4.2 Initialisierung

Bevor ein Objekt benutzt werden kann, muss ihm ein Wert übergeben werden. C++ bietet eine Vielzahl an Notationen zum Ausdrücken einer Initialisierung, wie etwa das bereits vorgestellte **=** sowie eine universelle Form, nämlich Listen, die durch ein Paar geschweifte Klammern (**{}**) begrenzt werden:

```
double d1 = 2.3;              // Initialisiert d1 auf 2.3
double d2 {2.3};              // Initialisiert d2 auf 2.3
double d3 = {2.3};            // Initialisiert d3 auf 2.3 (das = ist mit
                              // { ... } optional)

complex<double> z = 1;        // eine komplexe Zahl mit
                              // Gleitkomma-Skalaren doppelter
Genauigkeit
complex<double> z2 {d1, d2};
complex<double> z3 = {d1, d2};    // das = ist mit { ... } optional

vector<int> v {1, 2, 3, 4, 5, 6}; // ein Vektor aus ints
```

Die Art, den Operator **=** zu verwenden, ist traditionell und stammt schon aus C. Falls Sie sich unsicher sind, benutzen Sie die allgemeine **{}**-Listenform. Damit schützen Sie sich zumindest vor Konvertierungen, bei denen Informationen verloren gehen:

```
int i1 = 7.8;         // i1 wird zu 7 (Überraschung?!)
int i2 {7.8};         // Fehler: Gleitkomma- zu Integer-Konvertierung
```

Leider sind Konvertierungen, bei denen Informationen verloren gehen, die sogenannten *verengenden Konvertierungen* (Narrowing Conversions), wie etwa **`double`** nach **`int`** und **`int`** nach **`char`**, erlaubt und kommen implizit zum Einsatz, wenn Sie **=** benutzen (nicht jedoch, wenn Sie **{}** verwenden). Die durch implizite verengende Konvertierungen verursachten Probleme sind der Preis, den man für die Kompatibilität zu C bezahlen muss (§19.3).

Eine Konstante (§1.6) kann nicht uninitialisiert bleiben und eine Variable sollte nur unter ausgesprochen seltenen Umständen uninitialisiert bleiben. Führen Sie einen Namen nur dann ein, wenn Sie einen passenden Wert dafür haben. Benutzerdefinierte Typen (wie **`string`**, **`vector`**, **`Matrix`**, **`Motor_controller`** und **`Orc_warrior`**) können so definiert werden, dass sie implizit initialisiert werden (§5.2.1).

Wenn Sie eine Variable definieren, müssen Sie deren Typ nicht explizit angeben, falls der Typ sich aus der Initialisierung schlussfolgern lässt:

```
auto b = true;       // ein Boolean
auto ch = 'x';       // ein char
auto i = 123;        // ein int
auto d = 1.2;        // ein double
auto z = sqrt(y);    // z hat den Typ, der von sqrt(y) zurückgegeben wird
auto bb {true};      // bb ist ein Boolean
```

Bei **`auto`** neigt man meist dazu, **`=`** zu benutzen, weil keine potenziell lästige Typkonvertierung im Spiel ist. Falls Sie es jedoch vorziehen, konsistent die **`{}`**-Initialisierung zu verwenden, so können Sie das auch hier tun.

Man setzt **`auto`** immer dann ein, wenn es keinen speziellen Grund gibt, den Typ ausdrücklich zu erwähnen. »Spezielle Gründe«, den Typ dennoch anzugeben, sind unter anderem:

- Die Definition ist in einem großen Gültigkeitsbereich, in dem Sie den Leserinnen und Lesern Ihres Codes den Typ ganz eindeutig klarmachen wollen.
- Der Typ des Initialisierers ist nicht offensichtlich.
- Sie wollen den Umfang oder die Genauigkeit einer Variablen ganz ausdrücklich festlegen (z. B. **`double`** anstelle von **`float`**).

Durch Verwendung von **`auto`** werden Redundanz und das Schreiben langer Typnamen vermieden. Das ist vor allem in der generischen Programmierung wichtig, bei der es für die Programmierer schwierig sein kann, den exakten Typ eines Objekts zu kennen, und die Typnamen recht lang sein können (§13.2).

1.5 Gültigkeitsbereich und Lebensdauer

Eine Deklaration führt ihren Namen in einen Gültigkeitsbereich ein:

- *Lokaler Gültigkeitsbereich*: Ein Name, der in einer Funktion (§1.3) oder einem Lambda (§7.3.2) deklariert wurde, wird als *lokaler Name* bezeichnet. Sein Gültigkeitsbereich erstreckt sich vom Punkt der Deklaration bis zum Ende des Blocks, in dem seine Deklaration auftritt. Ein *Block* wird durch ein Paar

geschweifte Klammern (**{}**) begrenzt. Namen von Funktionsargumenten werden als lokale Namen betrachtet.

- *Klassen-Gültigkeitsbereich*: Ein Name wird als *Member-Name* (oder *Class-Member-Name*) bezeichnet, wenn er in einer Klasse (§2.2, §2.3, Kapitel 5), aber außerhalb einer Funktion (§1.3), eines Lambda (§7.3.2) oder eines **enum class** (§2.4) definiert wurde. Sein Gültigkeitsbereich erstreckt sich von der öffnenden geschweiften Klammer (**{**) seiner umschließenden Deklaration bis zur dazugehörenden schließenden geschweiften Klammer (**}**).
- *Namensraum-Gültigkeitsbereich*: Ein Name wird als *Namensraum-Member-Name* bezeichnet, wenn er in einem Namensraum (Namespace) (§3.3), aber außerhalb einer Funktion, eines Lambda (§7.3.2), einer Klasse (§2.2, §2.3, Kapitel 5) oder eines **enum class** (§2.4) definiert wurde. Sein Gültigkeitsbereich erstreckt sich vom Punkt der Deklaration bis zum Ende seines Namensraums.

Ein Name, der nicht innerhalb eines anderen Konstrukts deklariert wurde, wird als *globaler Name* bezeichnet und liegt im globalen Namensraum.

Darüber hinaus gibt es Objekte ohne Namen, wie etwa temporäre Objekte und Objekte, die mithilfe von **new** (§5.2.2) erzeugt wurden. Zum Beispiel:

```
vector<int> vec;  // vec ist global (ein globaler Vektor aus Integern)

void fct(int arg) // fct ist global (benennt eine globale Funktion)
                  // arg ist lokal (benennt ein Integer-Argument)
{
     string motto {"Wer wagt, gewinnt"}; // motto ist lokal
     auto p = new Record{"Hume"};        // p zeigt auf ein unbenanntes
                                         // Record (erzeugt durch new)
    // ...
}
struct Record {
     string name; // name ist ein Member von Record (ein String-Member)
    // ...
};
```

Ein Objekt muss vor seiner Benutzung konstruiert (initialisiert) werden. Am Ende seines Gültigkeitsbereichs wird es zerstört. Für ein Namensraumobjekt ist das Ende des Programms der Punkt der Zerstörung. Für ein Member wird der Punkt der Zerstörung durch den Punkt der Zerstörung des Objekts festgelegt, dessen Member es ist. Ein Objekt, das durch **new** erzeugt wurde, »lebt« hingegen, bis es mittels **delete** zerstört wird (§5.2.2).

1.6 Konstanten

C++ unterstützt zwei Arten von *Unveränderlichkeit* (damit ist ein Objekt mit einem unveränderlichen Zustand gemeint):

- **`const`** bedeutet in etwa: »Ich verspreche, diesen Wert nicht zu verändern«. Dies wird vor allem benutzt, um Schnittstellen zu spezifizieren, damit Daten mithilfe von Zeigern und Referenzen an Funktionen übergeben werden können, ohne dass man befürchten muss, dass sie modifiziert werden. Der Compiler setzt das Versprechen durch, das von **`const`** gegeben wurde. Der Wert eines **`const`** kann zur Laufzeit berechnet werden.
- **`constexpr`** bedeutet in etwa: »wird zum Zeitpunkt des Kompilierens ausgewertet«. Dies wird vor allem dafür verwendet, um Konstanten festzulegen, um die Ablage von Daten in schreibgeschütztem Speicher zu ermöglichen (wo es unwahrscheinlich ist, dass diese beschädigt werden) und zu Performance-Zwecken. Der Wert eines **`constexpr`** muss vom Compiler berechnet werden.

Zum Beispiel:

```
constexpr int dmv = 17;  // dmv ist eine benannte Konstante
int var = 17;            // var ist keine Konstante
const double sqv = sqrt(var);   // sqv ist eine benannte Konstante,
                      // die möglicherweise zur Laufzeit berechnet wird

double sum(const vector<double>&);    // sum wird sein Argument nicht
                                      // modifizieren (§1.7)

vector<double> v {1.2, 3.4, 4.5};    // v ist keine Konstante
const double s1 = sum(v);    // Okay: sum(v) wird zur Laufzeit
                             // ausgewertet
constexpr double s2 = sum(v);    // Fehler: sum(v) ist kein konstanter
                                 // Ausdruck
```

Damit eine Funktion in einem konstanten Ausdruck verwendet werden kann, das heißt in einem Ausdruck, der vom Compiler ausgewertet wird, muss sie mit **`constexpr`** oder **`consteval`** definiert werden. Zum Beispiel:

```
constexpr double square(double x) { return x*x; }

constexpr double max1 = 1.4*square(17);    // Okay: 1.4*square(17) ist
                                // ein konstanter Ausdruck
constexpr double max2 = 1.4*square(var);    // Fehler: var ist keine
              // Konstante, weshalb square(var) auch keine Konstante ist
```

```
const double max3 = 1.4*square(var);    // Okay: kann zur Laufzeit
                                        // ausgewertet werden
```

Eine **constexpr**-Funktion kann durchaus für nichtkonstante Argumente verwendet werden, allerdings ist das Ergebnis dann kein konstanter Ausdruck. Der Aufruf einer **constexpr**-Funktion mit nichtkonstanten Argumenten ist in Kontexten erlaubt, die keine konstanten Ausdrücke verlangen. Auf diese Weise müssen Sie die gleiche Funktion nicht zweimal definieren: einmal für konstante Ausdrücke und einmal für Variablen. Wenn Sie wollen, dass eine Funktion nur für die Auswertung während des Kompilierens benutzt wird, deklarieren Sie sie mit **consteval** statt mit **constexpr**. Zum Beispiel:

```
consteval double square2(double x) { return x*x; }

constexpr double max1 = 1.4*square2(17);    // Okay: 1.4*square(17) ist
                                            // ein konstanter Ausdruck
const double max3 = 1.4*square2(var);       // Fehler: var ist keine
                                            // Konstante
```

Funktionen, die mit **constexpr** oder **consteval** deklariert werden, sind die C++-Version des Prinzips von reinen Funktionen. Sie können keine Nebeneffekte haben und können nur Informationen verwenden, die ihnen als Argumente übergeben wurden. Insbesondere können sie nichtlokale Variablen nicht modifizieren, aber sie können Schleifen haben und ihre eigenen lokalen Variablen benutzen. Zum Beispiel:

```
constexpr double nth(double x, int n)    // angenommen 0<=n
{
     double res = 1;
     int i = 0;
     while (i<n) {        // while-Schleife: macht etwas, solange
                                 // die Bedingung erfüllt ist (§1.7.1)
         res *= x;
         ++i;
     }
     return res;
}
```

An einigen Stellen verlangen die Regeln der Sprache den Einsatz von konstanten Ausdrücken (z. B. bei Array-Grenzen (§1.7), Case-Bezeichnern (§1.8), Template-Wertargumenten (§7.2) und Konstanten, die mit **constexpr** deklariert werden). In

anderen Fällen ist die Auswertung zum Zeitpunkt des Kompilierens aus Performance-Gründen wichtig. Unabhängig von Performance-Fragen ist die Unveränderlichkeit von Objekten eine wichtige Designentscheidung.

1.7 Zeiger, Arrays und Referenzen

Die grundlegendste Form der Sammlung (Collection) von Daten ist ein zusammenhängender Speicherbereich, der mit Elementen desselben Typs belegt ist, ein sogenanntes *Array*. Im Prinzip ist es das, was die Hardware bereitstellt. Ein Array aus Elementen des Typs **char** kann folgendermaßen deklariert werden:

```
char v[6];        // Array aus sechs Zeichen
```

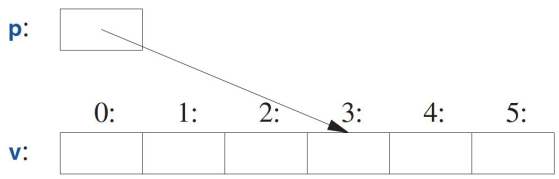
Ein Zeiger (Pointer) auf einen Speicherbereich lässt sich so deklarieren:

```
char* p;          // ein Zeiger auf ein Zeichen
```

In Deklarationen bedeutet **[]** »Array aus« und ***** »Zeiger auf«. Alle Arrays haben **0** als untere Grenze, sodass **v** die sechs Elemente **v[0]** bis **v[5]** besitzt. Die Größe eines Arrays muss ein konstanter Ausdruck sein (§1.6). Eine Zeigervariable kann die Adresse eines Objekts des entsprechenden Typs enthalten:

```
char* p = &v[3];    // p zeigt auf das vierte Element von v
char x = *p;        // *p ist das Objekt, auf das der Zeiger p zeigt
```

In einem Ausdruck bedeutet das unäre Präfix ***** »Inhalt von« und das unäre Präfix **&** »Adresse von«. Das kann grafisch so dargestellt werden:



Stellen Sie sich vor, Sie würden die Elemente eines Arrays ausgeben:

```
void print()
{
     int v1[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
    for (auto i=0; i!=10; ++i)     // gibt die Elemente aus
        cout << v[i] << '\n';
    // ...
}
```

Diese **for**-Anweisung kann gelesen werden als: »Setze **i** auf 0; solange **i** noch nicht **10** ist, gib das **i**-te Element aus und erhöhe **i** um 1«. C++ bietet auch eine einfachere **for**-Anweisung, die sogenannte bereichsbasierte **for**-Anweisung, für Schleifen, die eine Sequenz in der einfachsten Weise durchlaufen:

```
void print2()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto x : v)        // für jedes x in v
        cout << x << '\n';

    for (auto x : {10, 21, 32, 43, 54, 65})    // für jeden Integer in
                                               // der Liste
        cout << x << '\n';
    // ...
}
```

Die erste bereichsbasierte **for**-Anweisung kann gelesen werden als: »Kopiere jedes Element aus **v**, vom ersten bis zum letzten, nach **x** und gib es aus«. Beachten Sie, dass Sie keine Array-Grenze angeben müssen, wenn Sie es mit einer Liste initialisieren. Die bereichsbasierte **for**-Anweisung kann für jede Sequenz von Elementen benutzt werden (§13.1).

Falls Sie die Werte aus **v** nicht in die Variable **x** kopieren wollen, sondern mit **x** nur ein Element referenzieren möchten, könnten Sie schreiben:

```
void increment()
{
    int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    for (auto& x : v)    // addiert 1 zu jedem x in v
        ++x;
    // ...
}
```

In einer Deklaration bedeutet das unäre Suffix **&** »Referenz auf«. Eine Referenz ist vergleichbar mit einem Zeiger, allerdings müssen Sie nicht das Präfix * benutzen,