

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

Inhaltsverzeichnis

	Einleitung	11
	Danksagungen	13
	Über die Fachkorrektore der deutschen Ausgabe	13
1	Die Grundlagen	15
1.1	Einführung	15
1.2	Programme	15
1.3	Funktionen	18
1.4	Typen, Variablen und Arithmetik	21
1.5	Gültigkeitsbereich und Lebensdauer	25
1.6	Konstanten	27
1.7	Zeiger, Arrays und Referenzen	29
1.8	Bedingungen prüfen	33
1.9	Auf Hardware abbilden	36
1.10	Ratschläge	39
2	Benutzerdefinierte Typen	41
2.1	Einführung	41
2.2	Strukturen	42
2.3	Klassen	44
2.4	Aufzählungen	46
2.5	Unions	48
2.6	Ratschläge	51
3	Modularität	53
3.1	Einführung	53
3.2	Separates Kompilieren	54
3.3	Namensräume	62
3.4	Funktionsargumente und Rückgabewerte	64
3.5	Ratschläge	71
4	Fehlerbehandlung	73
4.1	Einführung	73
4.2	Exceptions	73
4.3	Invarianten	75
4.4	Alternativen für die Fehlerbehandlung	78

4.5	Assertions	81
4.6	Ratschläge	84
5	Klassen	87
5.1	Einführung	87
5.2	Konkrete Typen	88
5.3	Abstrakte Typen	96
5.4	Virtuelle Funktionen	100
5.5	Klassenhierarchien	101
5.6	Ratschläge	110
6	Notwendige Operationen	113
6.1	Einführung	113
6.2	Kopieren und Verschieben	117
6.3	Ressourcenverwaltung	123
6.4	Operatoren überladen	125
6.5	Konventionelle Operationen	126
6.6	Benutzerdefinierte Literale	131
6.7	Ratschläge	132
7	Templates	135
7.1	Einführung	135
7.2	Parametrisierte Typen	135
7.3	Parametrisierte Operationen	142
7.4	Template-Mechanismen	151
7.5	Ratschläge	156
8	Konzepte und generische Programmierung	157
8.1	Einführung	157
8.2	Konzepte	158
8.3	Generische Programmierung	169
8.4	Variadische Templates	173
8.5	Modell der Template-Kompilierung	176
8.6	Ratschläge	177
9	Überblick über die Bibliothek	179
9.1	Einführung	179
9.2	Komponenten der Standardbibliothek	180
9.3	Organisation der Standardbibliothek	181
9.4	Ratschläge	186

10	Strings und reguläre Ausdrücke	187
10.1	Einführung	187
10.2	Strings	187
10.3	String-Views	191
10.4	Reguläre Ausdrücke	193
10.5	Ratschläge	201
11	Eingabe und Ausgabe	203
11.1	Einführung	203
11.2	Ausgabe	204
11.3	Eingabe	205
11.4	I/O-Status	207
11.5	Ein-/Ausgabe benutzerdefinierter Typen	208
11.6	Ausgabeformatierung	210
11.7	Streams	216
11.8	Ein-/Ausgaben im C-Stil	220
11.9	Dateisystem	221
11.10	Ratschläge	226
12	Container	229
12.1	Einführung	229
12.2	vector	229
12.3	list	236
12.4	forward_list	238
12.5	map	238
12.6	unordered_map	240
12.7	Allokatoren	242
12.8	Ein Überblick über Container	244
12.9	Ratschläge	246
13	Algorithmen	249
13.1	Einführung	249
13.2	Verwendung von Iteratoren	252
13.3	Iterator-Typen	255
13.4	Verwendung von Prädikaten	260
13.5	Überblick über Algorithmen	260
13.6	Parallele Algorithmen	262
13.7	Ratschläge	264
14	Bereiche (Ranges)	265
14.1	Einführung	265

14.2	Views	266
14.3	Generatoren	269
14.4	Pipelines	270
14.5	Überblick über Konzepte.	271
14.6	Ratschläge.	278
15	Zeiger und Container	279
15.1	Einführung.	279
15.2	Zeiger	280
15.3	Container	287
15.4	Alternativen	297
15.5	Ratschläge.	302
16	Utilities	305
16.1	Einführung.	305
16.2	Zeit	305
16.3	Funktionsanpassung	309
16.4	Typfunktionen	310
16.5	source_location	317
16.6	move() und forward().	318
16.7	Bitmanipulation	320
16.8	Ein Programm beenden	321
16.9	Ratschläge.	322
17	Numerik	325
17.1	Einführung.	325
17.2	Mathematische Funktionen	325
17.3	Numerische Algorithmen	327
17.4	Komplexe Zahlen.	329
17.5	Zufallszahlen	330
17.6	Vektorarithmetik	333
17.7	Numerische Grenzen	333
17.8	Typ-Aliasse	334
17.9	Mathematische Konstanten.	334
17.10	Ratschläge.	335
18	Nebenläufigkeit	337
18.1	Einführung.	337
18.2	Tasks und thread	338
18.3	Daten gemeinsam nutzen	342

18.4	Warten auf Ereignisse	345
18.5	Kommunizierende Tasks	347
18.6	Koroutinen	354
18.7	Ratschläge	359
19	Geschichte und Kompatibilität	363
19.1	Geschichte	363
19.2	Die Entwicklung der Merkmale von C++	375
19.3	C/C++-Kompatibilität	381
19.4	Ratschläge	386
A	module std	389
B	Literaturverzeichnis	393
	Stichwortverzeichnis	399

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 weitermachen¹. 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.

Die Grundlagen

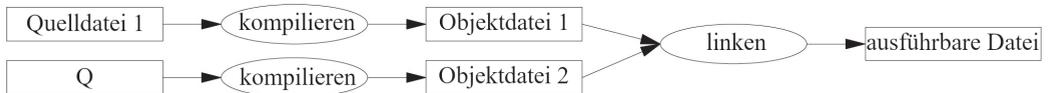
*The first thing we do, let's
kill all the language lawyers.
– Henry VI, Part II*

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 »Rückgabety« `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 Rückgabetyt 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 Compilezeitpunkt 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 Rückgabetyt, 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-Variablen.

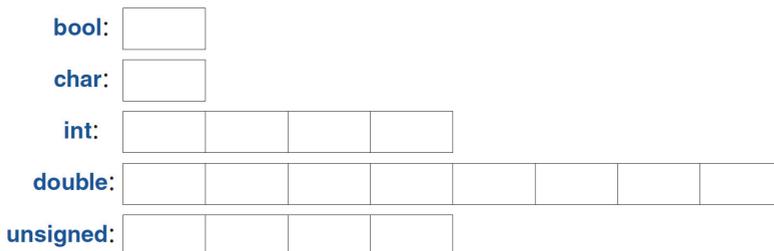
Eine Deklaration ist eine Anweisung, die eine Entität in das Programm einführt und ihren Typ festlegt:

- 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 auflisten 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 hexadezimal (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 $x.y$, $x \rightarrow y$, $x(y)$, $x[y]$, $x << y$, $x >> y$, $x \&\& y$ 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 uninitialized bleiben und eine Variable sollte nur unter ausgesprochen seltenen Umständen uninitialized 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 **constexpr** 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 **constexpr** statt mit **constexpr**. Zum Beispiel:

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

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

Funktionen, die mit **constexpr** oder **constexpr** 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
```

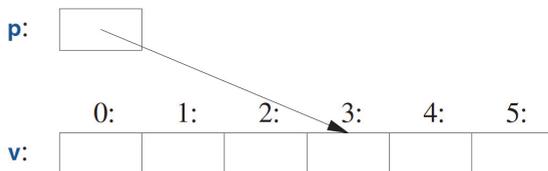
Ein Zeiger (Pointer) auf einen Speicherbereich lässt sich so deklarieren:

```
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,

um auf den Wert zuzugreifen, auf den durch die Referenz verwiesen wird. Außerdem kann im Gegensatz zu einem Zeiger eine Referenz nicht dazu gebracht werden, nach ihrer Initialisierung auf ein anderes Objekt zu verweisen.

Referenzen sind besonders nützlich beim Festlegen von Funktionsargumenten. Zum Beispiel:

```
void sort(vector<double>& v); // sortiert v (v ist ein Vektor aus doubles)
```

Indem Sie eine Referenz benutzen, stellen Sie sicher, dass Sie bei einem Aufruf von `sort(my_vec)` nicht aus Versehen `my_vec` kopieren. Es wird daher tatsächlich `my_vec` sortiert und nicht eine Kopie davon.

Wenn Sie ein Argument nicht modifizieren möchten, aber auch die Kosten des Kopierens vermeiden wollen, benutzen Sie eine `const`-Referenz (§1.6), das heißt eine Referenz auf eine Konstante. Zum Beispiel

```
double sum(const vector<double>&)
```

Funktionen, die `const`-Referenzen übernehmen, sind sehr verbreitet.

In Deklarationen werden Operatoren (wie `&`, `*` und `[]`) als *Deklaratoroperatoren* bezeichnet:

```
T a[n]    // T[n]: a ist ein Array aus n Ts
T* p      // T*: p ist ein Zeiger auf T
T& r      // T&: r ist eine Referenz auf T
T f(A)    // T(A): f ist eine Funktion, die ein Argument des Typs A
           // entgegennimmt und ein Ergebnis des Typs T zurückgibt
```

1.7.1 Der Null-Pointer

Sie versuchen, dafür zu sorgen, dass ein Zeiger immer auf ein Objekt zeigt, damit beim Dereferenzieren des Zeigers ein gültiges Ergebnis erzeugt wird. Wenn Sie kein Objekt haben, auf das gezeigt wird, oder Sie die Vorstellung vermitteln müssen, dass »kein Objekt verfügbar« ist (z. B. für das Ende einer Liste), dann geben Sie dem Zeiger den Wert `nullptr` (Null-Pointer). Es gibt für alle Typen von Zeigern nur ein `nullptr`:

```
double* pd = nullptr;
Link<Record>* lst = nullptr; // Zeiger auf einen Link auf ein Record
int x = nullptr;           // Fehler: nullptr ist ein Zeiger, kein Integer
```

Oft erweist es sich als klug zu überprüfen, dass ein Zeigerargument tatsächlich auf etwas zeigt:

```
int count_x(const char* p, char x)
// zählt, wie oft x in p[] auftritt
// p soll auf ein 0-terminiertes Array aus char (oder auf nichts) zeigen
{
    if (p==nullptr)
        return 0;
    int count = 0;
    for (; *p!=0; ++p)
        if (*p==x)
            ++count;
    return count;
}
```

Man kann mit `++` einen Zeiger weiterrücken, sodass er auf das nächste Element eines Arrays zeigt, und außerdem den Initialisierer in einer **for**-Anweisung weglassen, wenn er nicht gebraucht wird.

Die Definition `count_x()` geht davon aus, dass `char*` ein *String im Stil von C* ist, das heißt, dass der Zeiger auf ein 0-terminiertes Array aus `char` zeigt. Die Zeichen in einem String-Literal sind unveränderlich, um also `count_x("Hello!")` zu verarbeiten, habe ich `count_x()` mit einem `const char*`-Argument deklariert.

In älterem Code wird anstelle von `nullptr` üblicherweise `0` oder `NULL` benutzt. Mit `nullptr` vermeidet man jedoch eine mögliche Verwechslung zwischen Integern (wie `0` oder `NULL`) und Zeigern (wie `nullptr`).

In dem `count_x()`-Beispiel benutze ich für die **for**-Anweisung keinen Initialisierer, Sie können also auch die einfachere **while**-Anweisung einsetzen:

```
int count_x(const char* p, char x)
// zählt, wie oft x in p[] auftritt
// p soll auf ein 0-terminiertes Array aus char (oder auf nichts) zeigen
{
    if (p==nullptr)
        return 0;
    int count = 0;
    while (*p) {
        if (*p==x)
            ++count;
        ++p;
    }
    return count;
}
```

Die **while**-Anweisung wird ausgeführt, bis ihre Bedingung **false** wird.

Das Prüfen eines numerischen Werts (z. B. **while (*p)** in **count_x()**) ist äquivalent mit dem Vergleich des Werts mit **0** (hier also **while (*p!=0)**). Das Prüfen eines Zeigerwerts (z. B. **if (p)**) ist äquivalent mit dem Vergleich des Werts mit dem **nullptr** (hier also **if (p!=nullptr)**).

Es gibt keine »Nullreferenz«. Eine Referenz muss auf ein gültiges Objekt verweisen (und die Implementierungen gehen davon aus, dass sie es tut). Es gibt obstrukture und schlaue Möglichkeiten, diese Regel zu verletzen – machen Sie das nicht!

1.8 Bedingungen prüfen

C++ stellt die übliche Menge an Anweisungen zum Ausdrücken von Verzweigungen und Schleifen bereit, wie etwa **if**-Anweisungen, **switch**-Anweisungen, **while**- und **for**-Schleifen. Hier ist zum Beispiel eine einfache Funktion, die eine Benutzereingabe anfordert und einen booleschen Wert zurückgibt, der die Antwort anzeigt:

```
bool accept()
{
    cout << "Wollen Sie weitermachen (j oder n)?\n"; // Frage anzeigen
    char answer = 0; // auf einen Wert initialisieren, der nicht in
                    // der Eingabe auftaucht
    cin >> answer; // Antwort lesen

    if (answer == 'j')
        return true;
    return false;
}
```

Passend zum Ausgabeoperator **<<** (»ausgeben an«) gibt es den **>>**-Operator (»einlesen von«) für Eingaben; **cin** ist der Standard-Eingabe-Stream (Kapitel 11). Der Typ des rechten Operanden von **>>** legt fest, welche Eingabe akzeptiert wird; der rechte Operand ist außerdem das Ziel der Eingabeoperation. Das Zeichen **\n** am Ende des Ausgabestrings repräsentiert ein Newline (§1.2.1).

Beachten Sie, dass die Definition von **answer** dort erscheint, wo sie benötigt wird (und nicht schon vorher). Eine Deklaration kann überall dort auftauchen, wo auch eine Anweisung stehen kann.

Das Beispiel lässt sich noch verbessern, indem man auch die Antwort **n** (für »nein«) berücksichtigt:

```

bool accept2()
{
    cout << "Wollen Sie weitermachen (j oder n)?\n"; // Frage anzeigen
    char answer = 0; // auf einen Wert initialisieren, der nicht in
                    // der Eingabe auftaucht
    cin >> answer; // Antwort lesen

    switch (answer) {
    case 'j':
        return true;
    case 'n':
        return false;
    default:
        cout << "Das ist dann wohl ein Nein.\n";
        return false;
    }
}

```

Eine **switch**-Anweisung prüft einen Wert gegen eine Menge aus Konstanten. Diese Konstanten, **case**-Bezeichner genannt, müssen eindeutig sein. Falls der geprüfte Wert zu keinem der Bezeichner passt, wird **default** gewählt. Gibt es keinen zu dem Wert passenden **case**-Bezeichner und wurde auch kein **default** angegeben, dann findet gar keine Aktion statt.

Sie müssen ein **case** nicht verlassen, indem Sie aus der Funktion zurückkehren, die seine **switch**-Anweisung enthält. Oft wollen Sie die Ausführung mit der Anweisung fortsetzen, die auf die **switch**-Anweisung folgt. Das können Sie mit einer **break**-Anweisung erreichen. Schauen Sie sich als Beispiel einen überaus geschickten, wenn auch simplen Parser für ein einfaches, befehlsgesteuertes Videospiel an:

```

void action()
{
    while (true) {
        cout << "Aktion eingeben:\n"; // Aktion anfordern
        string act;
        cin >> act; // Zeichen in einen String einlesen
        Point delta {0,0}; // Point enthält ein {x,y} Paar

        for (char ch : act) {
            switch (ch) {
                case 'u': // nach oben (up)

```

```

        case 'n':                // nach Norden
            ++delta.y;
            break;
        case 'r':                // nach rechts
        case 'e':                // nach Osten (east)
            ++delta.x;
            break;
    // ... weitere Aktionen ...
    default:
        cout << "Ich hänge fest!\n";
    }
    move(current+delta*scale);
    update_display();
}
}
}

```

Genau wie eine **for**-Anweisung (§1.7) kann eine **if**-Anweisung eine Variable einführen und prüfen. Zum Beispiel:

```

void do_something(vector<int>& v)
{
    if (auto n = v.size(); n!=0) {
        // ... wir kommen hierher, falls n!=0 ...
    }
    // ...
}

```

Hier wird der Integer **n** für die Verwendung in der **if**-Anweisung definiert, mit **v.size()** initialisiert und sofort mit der Bedingung **n!=0** hinter dem Semikolon geprüft. Ein Name, der in einer Bedingung deklariert wird, befindet sich im Gültigkeitsbereich beider Zweige der **if**-Anweisung.

Wie bei der **for**-Anweisung deklariert man einen Namen in der Bedingung einer **if**-Anweisung, um den Gültigkeitsbereich der Variablen zu beschränken, womit man die Lesbarkeit verbessert und Fehler minimiert.

Am gebräuchlichsten ist es, eine Variable gegen **0** (oder den **nullptr**) zu prüfen. Dazu verzichten Sie einfach auf die explizite Erwähnung der Bedingung. Zum Beispiel:

```

void do_something(vector<int>& v)
{

```

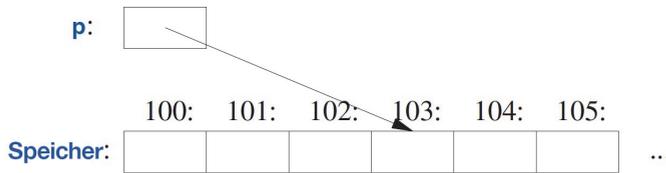
```
if (auto n = v.size()) {  
    // ... wir kommen hierher, falls n!=0 ...  
}  
// ...  
}
```

Nutzen Sie nach Möglichkeit immer diese knappere und einfachere Form.

1.9 Auf Hardware abbilden

C++ ermöglicht eine direkte Abbildung auf Hardware. Wenn Sie eine der grundlegenden Operationen benutzen, dann wird Ihre Implementierung die sein, die die Hardware bietet. Zum Beispiel führt das Addieren zweier **int**, **x+y**, direkt den entsprechenden Maschinenbefehl aus.

Eine C++-Implementierung sieht den Speicher eines Computers als eine Sequenz von Speicherorten, in die sie (typisierte) Objekte legen kann, die sie mithilfe von Zeigern adressiert, also anspricht:



Ein Zeiger wird im Speicher als eine Maschinenadresse dargestellt, der numerische Wert von **p** in dieser Abbildung wäre also **103**. Falls das verdächtig nach einem Array (§1.7) aussieht, dann liegt das daran, dass ein Array für C++ die grundsätzliche Abstraktion einer »fortlaufenden Abfolge von Objekten im Speicher« ist.

Die einfache Abbildung grundlegender Sprachkonstrukte auf die Hardware ist entscheidend für die sagenhafte, maschinennahe Arbeitsweise, für die C und C++ seit Jahrzehnten berühmt sind. Das C und C++ zugrunde liegende Maschinenmodell basiert tatsächlich auf der Computerhardware und nicht auf irgendwelchen Formen von Mathematik.

1.9.1 Zuweisung

Eine Zuweisung eines integrierten Typs ist ein einfacher Kopierbefehl auf Maschinenebene. Nehmen Sie dies hier an:

```
int x = 2;
int y = 3;
x = y;      // x wird 3; wir erhalten also x==y
```

Das ist eindeutig. Grafisch kann das so dargestellt werden:



Die zwei Objekte sind unabhängig voneinander. Sie können den Wert von **y** ändern, ohne dass der Wert von **x** beeinträchtigt wird. Zum Beispiel verändert **x=99** den Wert von **y** nicht. Anders als in Java, C# und anderen Sprachen – aber genauso wie in C – gilt das für alle Typen, nicht nur für **ints**.

Falls Sie wollen, dass unterschiedliche Objekte auf den gleichen (gemeinsam genutzten) Wert verweisen, dann müssen Sie das angeben. Zum Beispiel:

```
int x = 2;
int y = 3;
int* p = &x;
int* q = &y; // p!=q und *p!=*q
p = q;      // p wird &y; jetzt ist p==q, also (offensichtlich) *p==*q
```

Grafisch wird das so ausgedrückt:

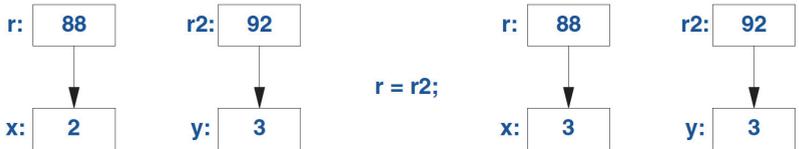


Ich habe willkürlich **88** und **92** als Adressen gewählt. Wieder können Sie sehen, dass das Objekt, auf das die Zuweisung erfolgt, den Wert aus dem zugewiesenen Objekt erhält, wodurch sich zwei unabhängige Objekte (hier: Zeiger) mit demselben Wert ergeben. Das heißt, **p=q** ergibt **p==q**. Nach **p=q** zeigen beide Zeiger auf **y**.

Eine Referenz und ein Zeiger verweisen/zeigen beide auf ein Objekt und beide werden im Speicher als Maschinenadresse dargestellt. Allerdings folgen sie unterschiedlichen Regeln in der Sprache. Die Zuweisung auf eine Referenz verändert nicht, worauf die Referenz verweist, sondern ändert den Inhalt des referenzierten Objekts:

```
int x = 2;  
int y = 3;  
int& r = x;    // r verweist auf x  
int& r2 = y;   // r2 verweist auf y  
r = r2;       // liest aus r2, schreibt auf r: x wird zu 3
```

Grafisch kann es so dargestellt werden:



Um auf den Wert zuzugreifen, auf den mit einem Zeiger gezeigt wird, benutzen Sie `*`; das wird für eine Referenz implizit gemacht.

Neben `x=y` haben Sie `x==y` für jeden integrierten Typ und gut designten benutzerdefinierten Typ (Kapitel 1), der `=` (Zuweisung) und `==` (Prüfung auf Gleichheit) anbietet.

1.9.2 Initialisierung

Die Initialisierung unterscheidet sich von der Zuweisung. Damit eine Zuweisung korrekt funktioniert, muss das Objekt, dem etwas zugewiesen wird, schon einen Wert haben. Die Aufgabe der Initialisierung andererseits ist es, ein nichtinitialisiertes Stück Speicher zu einem gültigen Objekt zu machen. Für fast alle Typen ist nicht definiert, welche Wirkung es hat, wenn aus einer nichtinitialisierten Variablen gelesen oder auf sie geschrieben wird. Betrachten Sie die Referenzen:

```
int x = 7;  
int& r {x};    // bindet r an x (r verweist auf x)  
r = 7;        // Zuweisung auf das, worauf r verweist  
  
int& r2;       // Fehler: nichtinitialisierte Referenz  
r2 = 99;      // Zuweisung auf das, worauf r2 verweist
```

Glücklicherweise können Sie keine nichtinitialisierte Referenz haben; wäre es möglich, dann würde dieses `r2 = 99` die `99` zu irgendeinem un spezifizierten Speicherort zuweisen; die Folge wären falsche Ergebnisse oder ein Absturz.

Sie können `=` verwenden, um eine Referenz zu initialisieren, aber lassen Sie sich davon nicht verwirren. Zum Beispiel:

```
int& r = x;    // bindet r an x (r verweist auf x)
```

Das ist immer noch eine Initialisierung, die **r** an **x** bindet, und nicht irgendeine Form des Kopierens von Werten.

Die Unterscheidung zwischen Initialisierung und Zuweisung ist auch für viele benutzerdefinierte Typen ausgesprochen wichtig, etwa für **string** und **vector**, bei denen das Objekt, dem etwas zugewiesen wurde, eine Ressource besitzt, die irgendwann wieder freigegeben werden muss (§6.3).

Die grundlegende Semantik der Argumentübergabe und Funktionswertrückgabe ist die der Initialisierung (§3.4). Zum Beispiel erhalten Sie auf diese Weise Referenzparameter (§3.4.1).

1.10 Ratschläge

Die Ratschläge, die Sie hier sehen, sind den C++ Core Guidelines [Stroustrup, 2015]¹ entnommen. Verweise auf Guidelines sehen so aus: [CG: ES.23], womit die 23. Regel im *Expressions and Statement* gemeint wäre. Im Allgemeinen bietet eine solche Core Guideline weitere Erklärungen und Beispiele.

- [1] Keine Panik! Alles wird im Laufe der Zeit klarer; §1.1; [CG: In.0].
- [2] Benutzen Sie nicht ausschließlich die integrierten Features. Viele grundlegende (integrierte) Features verwendet man normalerweise am besten indirekt durch Bibliotheken, wie die ISO-C++-Standardbibliothek (Kapitel 9–Kapitel 18); [CG: P.13].
- [3] Binden Sie die benötigten Bibliotheken mit **#include** oder (vorzugsweise) **import** ein, um das Programmieren zu vereinfachen; §1.2.1.
- [4] Sie müssen nicht jede Einzelheit von C++ kennen, um gute Programme zu schreiben.
- [5] Konzentrieren Sie sich auf Programmiertechniken, nicht auf Spracheigenschaften.
- [6] Der ISO-C++-Standard ist die entscheidende Instanz bei Fragen und Problemen zur Sprachdefinition; §19.1.3; [CG: P.2].
- [7] »Verpacken« Sie sinnvolle Operationen in Funktionen mit sorgfältig ausgewählten Namen; §1.3; [CG: F.1].
- [8] Eine Funktion sollte eine einzige logische Operation ausführen; §1.3; [CG: F.2].
- [9] Fassen Sie sich bei den Funktionen kurz; §1.3; [CG: F.3].
- [10] Verwenden Sie das Überladen, wenn Funktionen konzeptuell die gleiche Aufgabe mit unterschiedlichen Typen ausführen; §1.3.
- [11] Falls eine Funktion zum Zeitpunkt des Kompilierens ausgewertet werden darf, deklarieren Sie sie mit **constexpr**; §1.6; [CG: F.4].
- [12] Falls eine Funktion zum Zeitpunkt des Kompilierens ausgewertet werden muss, deklarieren Sie sie mit **constexpr**; §1.6.

1 <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>

- [13] Falls eine Funktion keine Nebeneffekte haben darf, deklarieren Sie sie mit **constexpr** oder **constexpr**; §1.6; [CG: F.4].
- [14] Entwickeln Sie Verständnis dafür, wie die Grundelemente der Sprache auf die Hardware abgebildet werden; §1.4, §1.7, §1.9, §2.3, §5.2.2, §5.4.
- [15] Nutzen Sie Trennzeichen, um große Literale besser lesbar zu machen; §1.4; [CG: NL.11].
- [16] Vermeiden Sie komplizierte Ausdrücke; [CG: ES.40].
- [17] Vermeiden Sie verengende Konvertierungen; §1.4.2; [CG: ES.46].
- [18] Minimieren Sie den Gültigkeitsbereich einer Variablen; §1.5, §1.8.
- [19] Halten Sie die Gültigkeitsbereiche klein; §1.5; [CG: ES.5].
- [20] Vermeiden Sie »magische Konstanten«, verwenden Sie symbolische Konstanten; §1.6; [CG: ES.45].
- [21] Bevorzugen Sie unveränderliche Daten; §1.6; [CG: P.10].
- [22] Deklarieren Sie (nur) einen Namen pro Deklaration; [CG: ES.10].
- [23] Halten Sie häufig benutzte und lokale Namen (Bezeichner) kurz; halten Sie selten benutzte und nichtlokale Namen länger; [CG: ES.7].
- [24] Vermeiden Sie ähnlich aussehende Namen; [CG: ES.8].
- [25] Vermeiden Sie **ALL_CAPS**-Namen; [CG: ES.9].
- [26] Benutzen Sie **auto**, um sich wiederholende Typnamen zu vermeiden; §1.4.2; [CG: ES.11].
- [27] Vermeiden Sie nichtinitialisierte Variablen; §1.4; [CG: ES.20].
- [28] Deklarieren Sie eine Variable erst, wenn Sie einen Wert haben, mit dem Sie sie initialisieren können; §1.7, §1.8; [CG: ES.21].
- [29] Wenn Sie eine Variable in der Bedingung einer **if**-Anweisung deklarieren, dann bevorzugen Sie die Version mit dem impliziten Prüfen gegen **0** oder **nullptr**; §1.8.
- [30] Bevorzugen Sie bereichsbasierte **for**-Schleifen gegenüber **for**-Schleifen mit einer expliziten Schleifenvariablen; §1.7.
- [31] Benutzen Sie **unsigned** nur für Bit-Manipulationen; §1.4; [CG: ES.101][CG: ES.106].
- [32] Halten Sie die Verwendung von Zeigern einfach und unkompliziert; §1.7; [CG: ES.42].
- [33] Benutzen Sie **nullptr** statt **0** oder **NULL**; §1.7; [CG: ES.47].
- [34] Sagen Sie in Kommentaren nichts, was ganz klar in Code ausgedrückt werden kann; [CG: NL.1].
- [35] Drücken Sie in Kommentaren Ihre Absichten aus; [CG: NL.2].
- [36] Wahren Sie einen konsistenten Stil bei Ihren Einrückungen; [CG: NL.4].

Stichwortverzeichnis

^, Bitweises Exklusiv-Oder 23
\
--, Dekrementierung 23
-, Minus-Operator 22
!, Logisches Nicht (Negation) 23
!=, Ungleichheit 22, 188
., Punkt-Operator 43
..., Parameterdeklaration 173
.cpp-Datei 56
.h-Datei 56
' , einfaches Anführungszeichen 22
(), Anwendungsoperator 144
(), explizite Größe 231
[], Array 29
[&], Capture-Liste 146
{}, Gruppierung 16
{}, Listen 24
*, Dereferenzieren 253
*, Inhalt von unäres Präfix 29
*, Multiplizieren 22
/, Dividieren 22
&, Bitweises Und 23
&, unäres Präfix, Adresse von 29
&, unäres Suffix, Referenz auf 30
&&, Logisches Und 23
&&, Rvalue-Verweis 121
#include 54, 182, 390
% 215
%, Modulo 22
+, Plus 22
+, Plus-Operator 22
++, Inkrementierung 23
++, Iterieren 253
<, Kleiner als 22
<<, Ausgabe 17
<=, Kleiner als oder gleich 22
<=>, Raumschiff-Operator 127
<algorithm> 261
<bit> 320
<chrono> 305
<cmath> 325
<complex> 326
<concepts> 271

<filesystem> 221
<fstream> 217
<limits> 333
<numeric> 261, 327
<regex> 193
<sstream> 217
<stddef> 334
<stdint> 334
<thread> 338
<type_traits> 311
>, Größer als 22
>=, Größer als oder gleich 22
>>, Eingabe 33
|, Bitweises Oder 23
||, Logisches Oder 23
~, Bitweises Komplement 23
~, Komplement 93

A

abort() 79, 322
Abstrakte Klasse 97
Abstrakter Typ 96
Abstraktionsmechanismus 41
Adressraum 339
Algorithmus 250
 Ausführungsstrategie 263
 parallele Ausführung 262
 vektorierte Ausführung 262
alloc 230
Alternative
 any 297, 301
 optional 297, 300
 union 297
 variant 297
Argumenttyp 19
Argumentübergabe 19, 65
 pass-by-reference 65
 pass-by-value 65
Arithmetische Konvertierung 23
Arithmetische Operation 23
Array 29
 assoziatives 239
array vs. eingebautes Array 290

- array vs. vector 290
- assert() 82
- Assertion 81
 - statische 83
- Assoziatives Array 239
- async() 351
- at() 188
- Aufzählung 41, 46
- Ausdruck
 - regulärer 193
- Ausnahme
 - current_exception() 349
- Auswertungsreihenfolge 24
- auto 25, 148, 167
- B**
- back_inserter() 251
- bad_alloc 77
- begin() 130, 165, 252
- Benutzerdefinierter Typ 41
- Bereich (range) 172, 265
 - Fabrik 269
 - Generator 269
 - View 266
- Bereichsfehler 326
- Beschränktes Template 138
- Bindung
 - strukturierte 69
- Bitmanipulation 320
- Buffer 139
- C**
- C
 - Maschinenmodell 36
- C++
 - Abstraktionsmechanismen 41
 - C mit Klassen 364
 - generische Programmierung 169
 - Hardware 36
 - Inspiration für 363
 - ISO-C++-Standards 370
 - Klasse 88
 - Kompatibilität 383
 - Maschinenmodell 36
 - separates Kompilieren 54
 - Standardbibliothek 16
 - typisierte Sprache 16
 - veraltete Funktionsmerkmale 380
 - vs. C 381
- C++20
 - Module 58
- C++-Programm
 - Funktionen 18
 - Hello, World! 16
 - Portabilität 15
- case-Bezeichner 34
- Cast (Typkonvertierung) 96
- cat() 192
- cerr 204, 216
- cin 33, 205, 216
- class 47
- Class-Member-Name 26
- clog 216
- Compiler 263
- complex() 89, 90
- compose() 192
- condition_variable 346
- const 27
- consteval 27
- constexpr 27
- Container 92, 229, 287
 - Allokator 242
 - array 289
 - assoziierte Typen 317
 - Beinahe-Container 288
 - Bereichsüberprüfung 234
 - bitset 291
 - deque 244
 - Element 233
 - forward_list 244
 - initialisieren 94
 - Initialisierungslistenkonstruktor 94
 - Iterator 252
 - Iteratoren 129
 - kopieren 118
 - list 244
 - map 244
 - multimap 244
 - multiset 244
 - pair 292
 - push_back() 94
 - Sequenz 129
 - set 244
 - Standardcontainer 244
 - tuple 295
 - unordered_map 244
 - unordered_multimap 244
 - unordered_multiset 244
 - unordered_set 244
 - vector 229, 244
 - verschieben 120
- Container-Operation 129
- Copy Elision 68, 123
- cout 17, 204, 216

D

Data Race 220, 339
 Datei-Stream
 fstream 216
 ifstream 217
 ofstream 217
 Dateisystem 221
 decltype() 311
 Deduktionsanleitung 142, 299
 defaultfloat, Format 211
 Deklaration 21, 53
 Deklaratoroperator 31
 delete 26
 Destruktor 74, 92
 Dictionary 239
 directory_entry 222
 directory_iterator 222
 Discriminant 50
 Domänenfehler 326
 Dreivegevergleich 127
 duration 306

E

E/A-Status 207
 E/A-Stream-Bibliothek 203
 Element 250
 end() 130, 165, 252
 enum 47
 equal_range() 294
 Ereignis 345
 Exception 68, 74, 368
 bad_alloc 77
 bad_any_access 301
 Hierarchie 77
 length_error 76
 out_of_range 75, 234
 exit() 79
 exit(x) 322
 export 61

F

Fehlerbehandlung 322
 Alternativen zur 78
 Exception 74, 79
 Fehlercode 79
 Resource Acquisition Is Initialization (RAII) 75
 filesystem_error 222
 filter() 270
 finally() 150
 find 252
 fixed, Format 211

fold() 175
 Fold-Ausdruck 175
 for
 Anweisung 30
 bereichsbasiert 30, 230, 251
 for_each() 309
 format() 213
 format-Bibliothek 210
 Formatierung
 % 215
 allgemein 211
 Ausgabe 210
 fest 211
 Manipulatoren 210
 printf() 212
 wissenschaftlich 211
 Format-String 213
 FORMULA 22, 23, 115, 188
 forward_list 238
 forward() 320
 fstream 216
 Funktion 18
 (), Aufrufoperator 310
 Argumenttyp 19
 Argumentübergabe 65
 Aufruf 64
 Definition 53
 Definition vs. Deklaration 54
 Exception 68
 function 310
 Funktionsargument 309
 Informationsweitergabe 65
 Lambda-Ausdruck 309
 Mehrdeutigkeit 20
 mem_fn() 309
 Member-Funktion 19
 Rückgabety 18, 68
 strukturierte Bindung 69
 Typ 19
 Typfunktion 310
 Überladen 20
 virtuell 97, 100
 Werterückgabe 66
 Funktionsargument 309
 Funktionsaufruf 64
 Funktionsdeklaration 18
 Funktionsobjekt
 Policy-Objekte 146
 Funktionsobjekt (Funkt) 144
 future 348

G

Garbage Collection 124, 284

- Generische Programmierung 169
- Gleitkommazahl 22
- greedy Match 200
- Gültigkeitsbereich
 - Klassen 26
 - lokaler 25
 - Namensraum 26
- H**
- Handle-to-Data-Modell 94
- hash<> 131
- Header-Datei 54, 55
- Header vs. Module 59
- Heap 42, 93
- I**
- I/O-Stream-Bibliothek 203
- ifstream 217, 259
- Implementierungsvererbung 105
- import 17, 54, 182
- Indizierung 230
- Initialisierung 24, 38
 - { } 24
 - = 24
 - Unterschied zur Zuweisung 38
- Initialisierungslistenkonstruktor 94
- Inlining 90
- int* 280
- Integer 22
- Integrierter Typ 41
- Invariante 76
- iostream 207
- ISO-C++-Standard 16
- ISO-C++-Standards 370
- ispanstream 219
- istream 203
- istringstream 217
- Iteration 188
- Iterator 255
 - istream_iterator 257
 - Konzept 256
 - ostream_iterator 257
 - Typ 255
- Iterator-Modell 130
- J**
- join() 338
- K**
- Kalender 306
- Klasse 19, 41, 44, 88
 - Ableitung 101
 - abstrakt 97
 - Basis 98
 - Hierarchie 101
 - Implementierungsvererbung 105
 - Invariante 76
 - konkrete 88
 - Konstruktor 45
 - Member 44
 - overloaded 299
 - path 221
 - polymorpher Typ 97
 - private-Member 44
 - public-Member 44
 - Ressourcen-Handle 118
 - Schnittstelle 44
 - Schnittstellenvererbung 105
 - Subklasse 98
 - Superklasse 98
 - Vererbung 98
 - Zeiger-Member 114
- Klassenhierarchie 101
- Klassen-Invariante 76
- Konkrete Klasse 88
- Konstante 27
 - mathematische 334
- Konstruktor 45, 114
- Konvertierung 116
 - arithmetische 23
 - aus Argumenttyp 116
 - implizite 116
 - Operatoren 92
 - Typ 96
 - verengende 24
- Konzept 138, 158, 271
 - assignable_from 272
 - auto 167
 - Bereichskonzept 277
 - bidirectional_iterator 275
 - bidirectional_range 277
 - common_range 277
 - common_reference_with 272
 - common_with 272
 - constructible_from 273
 - contiguous_iterator 275
 - contiguous_range 277
 - convertible_to 272
 - copy_constructible 273
 - copyable 274
 - default_initializable 273
 - Definition 163
 - derived_from 271
 - destructible 273
 - equality_comparable 273

equality_comparable_with 273
 equivalence_relation 274
 floating_point 272
 forward_iterator 275
 forward_range 277
 generische Programmierung 169
 input_iterator 275
 input_or_output_iterator 275
 input_range 277
 integral 272
 invocable 274
 Iterator-Konzepte 275
 konzeptbasiertes Überladen 161
 mergeable 275
 movable 274
 move_constructible 273
 output_iterator 275
 output_range 277
 permutable 275
 predicate 274
 random_access_iterator 275
 random_access_range 277
 range 277
 regular 170, 274
 regular_invocable 274
 relation 274
 same_as 271
 semiregular 274
 sentinel_for 275
 signed_integral 272
 Single-Argument 169
 sized_range 277
 sized_sentinel_for 275
 sortable 275
 strict_weak_order 274
 swappable 272
 swappable_with 272
 three_way_comparable 273
 three_way_comparable_with 273
 totally_ordered 273
 totally_ordered_with 273
 Typfunktion 311
 Typkonzept 271
 und Typ 169
 unsigned_integral 272
 view 277
 Konzeptbasiertes Überladen 161
 Koroutine 354
 Ereigniswarteschlange 356
 Kontextwechsel 356
 kooperatives Multitasking 356
 Nachrichtenwarteschlange 356
 Polymorphismus 356

Scheduler 356

L

Lambda-Ausdruck 146, 262, 309
 als Funktionsargument 147
 für Initialisierung 148
 generisch 148
 Laufzeit-Scheduler 263
 lazy Match 200
 Lebensdauer 25
 Leftinks-Fold 175
 Leser-Schreiber-Sperre 344
 Lexikografische Ordnung 188
 Lifting 172
 list 236
 Liste 24
 Literal
 benutzerdefiniert 131
 Literal-Operator 132
 User-Defined Literals (UDL) 132

M

main() 16
 make_shared() 284
 map 238
 (Schlüssel,Wert)-Paar 260
 Dictionary 239
 Schlüssel 239
 Wert 239
 Mathematische Funktion
 pow() 330
 sqrt() 330
 Mathematische Funktionen
 <cmath> 325
 Mathematische Konstante 334
 Mehrfachvererbung 369
 mem_fn() 309
 Member 44
 Member-Funktion 19, 89
 Member-Name 26
 Memory Exhaustion 77
 Modul 54, 58
 Modularität 57
 module std
 C++23 389
 Header-Einheit importieren 392
 move() 122, 318
 mutex 342

N

n, Newline 17, 195, 207
 Namensraum 62, 182

- globaler 26
- ranges 183, 265
- std 182, 261, 306
- Namensraum-Member-Name 26
- namespace-Direktive 64
- Nebenläufigkeit
 - Argumente übergeben 340
 - async() 351
 - Data Race 339
 - Ereignis 345
 - Ergebnisse übergeben 341
 - future 348
 - Koroutine 354
 - mutex 342
 - osyncstream 339
 - packaged_task 350
 - promise 348
 - queue 346
 - Task 338
 - Thread 338
- new 26
- noexcept 80, 84
- Nullerregel 115
- Null-Pointer 31
 - nullptr 31
- nullptr 31
- O**
- Objekt 21, 114
 - Initialisierung 26
 - kopieren 117
 - Kopier-Konstruktor 118
 - Kopier-Zuweisung 118
 - Zerstörung 26
- ofstream 217, 259
- Operation
 - , Dekrementierung 23
 - !=, Ungleichheit 188
 - »is instance of« 107
 - »is kind of« 107
 - *=, Skalierung 23
 - /=, Skalierung 23
 - %= 23
 - ++, Inkrementierung 23
 - += 23, 159, 188
 - = 23
 - =, Zuweisung 188
 - ==, Gleichheit 188
 - arithmetische 23
 - at() 188
 - Container 129
 - Funktions-Objekt 143
 - Funktions-Template 143
 - konventionell 126
 - Lambda-Ausdruck 143
 - parametrisiert 143
 - to_string() 292
 - to_ullong() 292
 - Vergleich 127
 - Verkettung 187
- Operationen
 - hash 241
- Operator
 - ^, Bitweises Exklusiv-Oder 23
 - , Minus 22
 - !, Logisches Nicht (Negation) 23
 - !=, Ungleichheit 22
 - ., Punkt-Operator 43
 - () , Anwendungsoperator 144
 - *, Dereferenzieren 253
 - *, Multiplizieren 22
 - /, Dividieren 22
 - &, Bitweises Und 23
 - &&, Logisches Und 23
 - %, Modulo 22
 - +, Plus 22, 121
 - ++, Iterieren 253
 - <, Kleiner als 22
 - << 204
 - <<, Ausgabe 17
 - <=, Kleiner als oder gleich 22
 - <=>, Raumschiff-Operator 127
 - =, Zuweisung 23
 - ==, Gleichheit 22
 - > 163, 315
 - >, Größer als 22
 - >=, Größer als oder gleich 22
 - >> 205
 - >>, Eingabe 33
 - | 270
 - |, Bitweises Oder 23
 - ||, Logisches Oder 23
 - ~, Bitweises Komplement 23
 - ~, Komplement 93
 - benutzerdefiniert 92
 - Deklaratoroperatoren 31
 - delete 26, 93, 242
 - für benutzerdefinierte Typen 125
 - new 26, 242
 - sizeof 22
 - überladen 92, 125
- operator() 144
- Ordnung
 - lexikografische 188
- ospanstream 219
- ostream 203

- ostreamstream 217
- osyncstream 219, 339
- override 98
- P**
- packaged_task 350
- Paralleler numerischer Algorithmus
 - par 328
 - par_unseq 328
 - seq 328
 - unseq 328
- Parameterpaket 173
- path 221, 222
- Pipeline 270
- Policy-Objekt 146
- Polymorpher Typ 97
- Portabilität von C-Programmen 15
- Prädikat 145, 260
- precision() 211
- printf() 212, 220
- Programm beenden
 - abort() 322
 - exit(x) 322
 - quick_exit(x) 322
 - terminate() 322
- Programmierung
 - generische 169
 - prozedurale 15
- promise 348
- Prozedurale Programmierung 15
- push_back() 94, 231
- Q**
- Quelldatei 15
- quick_exit(x) 322
- R**
- range 159
 - Definition 265
- range_value_t 254
- ranges, Namensraum 183
- RechtsRight-Fold 175
- recursive_directory_iterator 222
- Referenz 29
- regex_iterator 200
- regex_replace() 194
- regex_search() 194
- regex_token_iterator 194
- Regulärer Ausdruck 193, 195
 - ^ 195
 - ? 195
 - . 195
 - (195
 -) 195
 -] 195
 - { 195
 - } 195
 - * 195
 - + 195
 - | 195
 - \$ 195
 - alnum 197
 - alpha 197
 - blank 197
 - cntrl 197
 - D 198
 - d 197, 198
 - digit 197
 - graph 197
 - greedy Match 200
 - L 198
 - l 198
 - lazy Match 200
 - lower 197
 - print 197
 - punct 197
 - regex_match() 194
 - regex_search() 194
 - S 198
 - s 197, 198
 - space 197
 - U 198
 - u 198
 - upper 197
 - W 198
 - w 197, 198
 - xdigit 197
- replace() 188
- requires-Ausdruck 162
- Resource Acquisition Is Initialization (RAII)
 - 75, 94, 124, 282, 368
- Ressource 281
 - Beispiele 124
 - Leck 108
- Ressourcen-Handle 89
 - thread 123
 - unique_ptr 123
- Ressourcen-Handles vs. smarte Zeiger 285
- Ressourcenleck 108
- Ressourcenverwaltung 123, 281
 - Garbage Collection 124
 - Invarianten 78
- Rückgabety 68
- Rule of Zero (Nullregel) 115

S

- scanf() 220
- Schleife 30
- Schnittstelle 369
 - Deklaration 53
- Schnittstellenvererbung 105
- scientific, Format 211
- scoped_lock 343
- Sentinel 276
- shared_ptr 280, 282
- Short-String-Optimierung 190
- SIMD (Single Instruction, Multiple Data) 262
- size() 230
- source_location 317
- span 285
- spanstream 219
- Speicher
 - dynamisch 42, 93, 242
 - Fragmentierung 190, 243
 - frei 42, 89
 - Handle-to-Data-Modell 94
 - Heap 42
 - polymorph 244
- Speichererschöpfung 77
- Speicher-Stream
 - ispanstream 219
 - ospanstream 219
 - spanstream 219
 - strstream 219
- Stack entladen 74
- Standardbibliothek 16, 369
 - <chrono> 305
 - <limits> 333
 - <random> 330
 - <regex> 193
 - <type_traits> 312
 - <valarray> 333
- abort() 322
- Algorithmus 260
- Bereichsadaptor 267
- Container 229
- C-Standardbibliothek 180
- Exception-Hierarchie 77
- exit(x) 322
- Fabrik 269
- forward_list 238
- Generator 269
- Hash-Funktion 240
- Header 184
- Komponenten 180
- Konzept 271
- list 236
- map 238
 - module std 389
 - Namensraum 182
 - Nebenläufigkeit 337
 - Organisation 181
 - Pipeline 270
 - quick_exit(x) 322
 - source_location 317
 - Sprachunterstützung zur Laufzeit 180
 - std: 321
 - string_view 191
 - terminate() 322
 - tuple 295
 - unique_ptr 108
 - View 267
- Standardkonstruktor 90
- Standard-Member-Initialisierer 116
- Standard-Stream 216
 - cerr 216
 - cin 216
 - clog 216
 - cout 216
- static_assert 83
- static_cast 96
- Statische Assertion 83
- std
 - Header 184
 - Sub-Namensraum 182
- std: 17
- std::byte 321
- std::regex 193
- Stream
 - Iterator 257
 - synchronisierter 219
- String
 - Addition 187
 - C-Stil 189
 - Implementierung 189
 - Indizierung 188
 - Iteration 188
 - lexikografische Ordnung 188
 - Literal 189
 - replace() 188
 - string_view 191
 - substr() 188
 - Substring-Operationen 188
 - Vergleich 189
 - Zeichensätze 190
- string_type 223
- string_view 191
- String-Stream
 - istringstream 217
 - ostreamstream 217
 - stringstream 217

stringstream 217
 ostream 219
 struct 42, 296
 Strukturierte Bindung 69
 Subklasse 98
 substr() 188
 Suffix-Rückgabety 69
 Superklasse 98
 swap() 131
 Synchronisierter Stream 219
 Data Race 220
 osyncstream 219
 system_clock 308

T

Tag 50
 tagged Union 50
 Template
 Abstraktion mittels 170
 Alias 153
 beschränkt 138, 158
 beschränktes Argument 138
 Compile-Zeit-if 154
 Compile-Zeit-Mechanismus 137
 Instanziierung 138
 Konzept 138
 Parametrisierung 135, 138
 Spezialisierung 138
 Template-Argument 164
 Template-Metaprogrammierung 312
 Typprüfung 177
 unbeschränkt 177
 Variablen-Template 152
 variadisch 173
 Wertargument 139
 Template-Argument 138
 Deduktion 140
 Instanziierung 138
 Spezialisierung 138
 terminate() 79, 322
 Text
 Manipulation 187
 Thread
 Adressraum 339
 Lock 339
 stoppen 352
 throw 74
 time_point 306
 time_zone 308
 Trennzeichen 22
 try-Block 75
 Typ 21
 abstrakt 96

benutzerdefiniert 41, 89, 208, 233
 integriert 41
 konkret 96
 numerisch 233
 parametrisiert 135
 polymorpher 97
 string_view 187
 Zeiger 233
 Typ-Alias 334
 Typdeduktion 295
 Typfunktion 310
 Metaprogrammierung 312
 Typgenerator 316
 Typprädiat 312
 Typgenerator 316
 Typkonvertierung (Cast) 96
 Typparameter 135
 Typsystem 73

U

Überladen 20
 Übersetzungseinheit 57
 Uhr 305
 Union 48
 unique_lock 347
 unique_ptr 108, 280, 282
 unordered_map 240
 Unveränderlichkeit 27
 using-Deklaration 63, 258
 using-Direktive 63

V

value_type 223
 Variable 21
 atomic 344
 char 22
 Variablen-Template 152
 Variadisches Template 173
 variant 50
 Verengende Konvertierung 24
 Vererbung 98
 Verschiebe-Konstruktor 121
 Verschieben vs. Kopieren 318
 Verschiebe-Zuweisung 122
 vformat() 215
 View 266
 virtual 97
 Virtuelle Funktion 97
 Virtuelle Funktionstabelle 100
 void 18
 vtbl 100

W

weak_ptr 280
Wert 21
what() 75
Whitespace 206

Y

year_month_day 307

Z

Zahl

 complex 329
 float 329
 Gleitkommazahl 22
 Integer 22
 komplex 329

Zahlen

 double 329

Zeiger 29, 280

 besitzend 280
 hängend 280
 int* 280
 integriert 280

Maschinenadresse 36

Null-Pointer 31

Ressourcen-Handles vs. smarte Zeiger
 285

shared_ptr 280, 282

smart 282

unique_ptr 108, 123, 280, 282

weak_ptr 280

Zeit

 <chrono> 305

 duration 306

 time_point 306

 time_zone 308

 Zeitzone 308

Zeitzone 308

Zufallszahl

 <random> 330

 Engine 330

 Seed-Wert 332

 Verteilung 330

Zufallszahlengenerator 330

Zuweisung 23, 36, 114