C++

Das umfassende Handbuch

Kapitel 1
Das C++-Handbuch

Mit diesem Buch auf Ihrem Schreibtisch (denn für Ihre Hände ist es wohl zu schwer) hoffe ich, Ihnen ein Werk zu liefern, dass für Sie die Brücke zwischen Lehrbuch und Referenz darstellt. Ich möchte Ihnen umfassend den Weg zum Programmieren mit C++ bis zu dem Punkt ebnen, an dem Sie wissen, wo Sie weiter nachschlagen können. Das ist vielleicht ein etwas seltsames Ansinnen, aber ich gehe hier bewusst zwei Kompromisse ein: Erstens hoffe ich, dass Sie schon ein wenig über Programmieren im Allgemeinen wissen und vielleicht schon erste Erfahrungen mit der »Denkweise« des Computers gesammelt haben. C++ selbst müssen Sie noch nicht unbedingt kennen, hier setzt dieses Buch auf. Zweitens gibt es zu jedem Sprachelement viele Details, Einsatzmöglichkeiten und Interaktionen mit anderen Sprachelementen – und von diesen *sehr, sehr* viele –, sodass ich Ihnen zwar jedes Sprachelement beschreibe, aber nur bis zu einer gewissen Tiefe. Ich bette die Beschreibungen aber in einen Kontext ein, der Ihnen dabei hilft, ein Verständnis zu entwickeln. Der reine Text des Sprachstandards von C++ inklusive der dazugehörigen Standardbibliothek umfasst über 2000 Seiten – eng gedruckt und formal aufgeschrieben (das Dokument »N4971« ohne Index). Ein Werk wie das vorliegende Buch kann nicht anders, als Ihnen diese auf etwa 1000 Seiten verständlich, aber umfassend aufzubereiten, um dann durch eine umfassende Referenz an anderer Stelle ergänzt zu werden. Ich empfehle Referenzseiten (*http://cppreference.com*), Foren (*http://stackoverflow.com*) und die Suche (*http://google.com*, *http://bing.com*) im Internet.

Ich habe dieses Buch so aufgebaut, dass Sie zunächst das Werkzeug, mit dem Sie arbeiten, besser kennenlernen werden. Sie bekommen also Antworten auf die Fragen, was ein Compiler bzw. eine Entwicklungsumgebung ist und wie Sie beides einrichten. Dann erhalten Sie einen schnellen Überblick aus der Vogelperspektive, damit Ihnen die Lektüre der späteren Kapitel leichter fällt.

Danach wird es ausführlicher. Zunächst lernen Sie den Sprachkern kennen: Wie ist ein C++-Programm aufgebaut, und was für Elemente enthält Ihr Code? Hier geht es also hauptsächlich um Syntax und Semantik, Sie sehen außerdem die eingebauten Datentypen und erfahren, wie der Computer mit ihnen rechnet.

Es folgt eine umfassende Beschreibung der Standardbibliothek mit all ihren Werkzeugen, aber auch den Konzepten, die für ihren effektiven Einsatz wichtig sind.

1.1 Neu und modern

Bei all dem lernen Sie durchgehend das *neue* C++ kennen. Fragen Sie jetzt: »Warum neu?«, dann antworte ich Ihnen: »Weil beginnend mit C++11 für C++ ein *neues* Zeitalter angefangen hat«. C++ ist runderneuert worden und wird es immer noch. Mit C++11, C++14, C++17, C++20 und dem neuen C++23 ist es in C++ möglich geworden, so zu programmieren, dass man verständlichere, fehlerfreiere und nachhaltigere Programme schreibt – wenn, ja *wenn*, man die Elemente richtig einsetzt.

Im englischen Sprachraum wird für die neue Art, in C++ zu programmieren, der Begriff *modern* verwendet. Im Deutschen passt er nicht so ganz und hat eine leicht andere und daher nicht ganz passende Konnotation. Ich nehme daher lieber *neu* als Begriff, den Sie in diesem Buch daher an der einen oder anderen Stelle finden werden.

Aber was macht diese *neue* oder *moderne* C++-Programmierung aus?

- Sie können kompakter programmieren, weil der Compiler Ihnen viel Ballast abnehmen kann. *Typinferenz* mit `auto` und das bereichsbasierte `for` sind Beispiele dazu.
- Sie schreiben sichereren Code, wenn Sie die *vereinheitlichte Initialisierung* mit {…} bevorzugen, weniger *rohe Zeiger* verwenden, die *Container* und *Algorithmen* der Standardbibliothek nutzen und nicht zuletzt RAII verinnerlichen (siehe Kapitel 17, das vierte der »Dan«-Kapitel über guten Code).
- Durch *Verschieben* statt Kopieren werden Sie effizienter, ohne etwas dafür tun zu müssen außer Dinge wegzulassen, siehe Kapitel 17.
- Neue C++-Konstrukte sind eine Alternative zu weniger sicheren C-Mitteln.

Wenn ich in diesem Buch Beispiele präsentiere, sollen sie instruktiv und sinnvoll sein; aber dennoch einfach, kurz und in der Buchform übersichtlich. Ganz wichtig ist mir Praxisnähe. Und da kann es passieren, dass ich das eine oder andere *Muster* verwende, das für das Beispiel vielleicht nicht nötig gewesen wäre. So wird Ihnen zum Beispiel das *Pimpl-Pattern* in Kapitel 11, »Guter Code, 2. Dan: Modularisierung«, begegnen. Im Unterschied zu Mustern wie RAII, die eher im neuen C++ wichtig sind, wird deren Erklärung meist knapp ausfallen, denn hier tendiert meine Abwägung zu »kurz«. Alle etablierten Patterns und Idiome gehören durchaus auch in ein Buch, im Fokus dieses Buchs sind jedoch die neuen.

1.2 »Dan«-Kapitel

An strategisch passenden Stellen habe ich besondere Kapitel eingebaut, die sich weniger mit C++ an sich beschäftigen, Ihnen bei der Programmierung mit C++ aber dennoch helfen werden. Die Themen sind allgemeingültiger Natur aus der Softwareent-

wicklung wie Modularisierung, Testen und Ressourcenmanagement, aber konkret angewandt im Kontext von C++.

Diese »Dan«-Kapitel stehen an Stellen im Buch, die thematisch zu den umliegenden Kapiteln gehören. Sie sind aber bewusst weit gefasst und bauen nicht nur auf den vorhergehenden Kapiteln auf. Sie haben mehr generellen Handbuchcharakter und laden dazu ein, sich auch später noch praxisnahe Tipps abzuholen. Beim sequenziellen Durcharbeiten ist der eine oder andere Vorgriff akzeptiert und durchaus gewollt.

Ein besonderes Augenmerk gilt Kapitel 29, »Threads – Programmieren mit Mehrläufigkeit«, das, wenn auch sehr kompakt, die meisten praktischen Tipps enthält.

1.3 Darstellung in diesem Buch

In diesem Buch verwende ich durchgehend C++11, C++14, C++17 und C++20 als Standard, wenn ich Ihnen Dinge erkläre. Alle neuen C++-Compiler unterstützen den Großteil der Features. Auf die meisten C++20-Features weise ich hin, denn Sie können in Ihrem Projekt vielleicht nicht den brandheißen neuesten Compiler einsetzen. Wenn Sie in einer Umgebung arbeiten, die einen wirklich alten Compiler bedingt, müssen Sie auf einige nützliche Dinge aus dem C++-Sprachkern verzichten und haben nur Zugriff auf einen eingeschränkten Teil der Standardbibliothek. Entnehmen Sie der Dokumentation Ihres alten Compilers, was Sie nutzen können, und schauen Sie bei *boost* nach, ob Ihre Standardbibliothek damit zu erweitern ist.

Dinge, die Sie erst in der neuesten C++-Version C++23 finden, werde ich im Text erwähnen und im Quellcode besonders markieren.

1.4 Verwendete Formatierungen

Listings enthalten die folgenden Elemente:

```
// https://godbolt.org/z/jrqEGvh1M
#include <iostream>                // cout
#include <memory>                  // make_shared
int main() {                       // ein Kommentar
  std::cout << "Blopp\n";          // hervorgehoben
  Typ feh-ler(args);               // ↯ Zeile mit einem Fehler
  if consteval {                   // auf C++23-Features weise ich hin
      sin(55); }
  for(;;) break; // andere Markierung, zur Unterscheidung oder Hervorhebung
}
```

Listing 1.1 Ein kleines Formatbeispiel

Kasten

Kästen enthalten wichtige Dinge, die Sie sich besonders merken sollten.

Balken

Mit einem Balken sind Einschübe markiert, die meist weitergehende Hinweise enthalten. Zu Beginn der meisten Kapitel finden Sie ein *Kapiteltelegramm* mit eingeführten Begriffen mit dieser Kennzeichnung.

Wenn ich einen Namen für eine Symbolfolge im Text verwende, bemühe ich mich, direkt dahinter auch die Symbolfolge anzugeben. Beim Lesen anderer Bücher hätte ich das manchmal hilfreich gefunden. Ich setze das Symbol dann nicht extra zwischen Zitatklammern »«, da ich das in vielen Fällen überladen und manchmal sogar verwirrend finde. Hier ein Beispiel: Sie schreiben Strings in doppelte Anführungszeichen ", verwenden eine Referenz &, nutzen runde Klammern () und spitze Klammern <> und setzen ein Komma zwischen Parameter.

Beispielmaterial zum Download

Unter *https://www.rheinwerk-verlag.de/5809* finden Sie die Programmbeispiele und Listings zum Herunterladen.

Wenn Sie Fragen oder Bemerkungen haben oder Ihnen eine Diskussion hilft, erreichen Sie mich auf *http://cpp11.generisch.de* oder per E-Mail an *torsten.t.will@gmail.com*.

1.5 Sorry for my Denglish

Wenn man übers Programmieren redet oder schreibt, entsteht zwangsläufig ein Konflikt darüber, ob man Englisch oder Deutsch oder beides verwenden sollte. Manche deutschen auf Teufel komm raus ein, was dann die unmöglichsten Wortkonstrukte ergibt. Ich erinnere mich noch an die Übersetzung des »Joysticks« in der Anleitung eines Computerspiels ...

Ich persönlich bevorzuge das andere Extrem und bleibe lieber beim englischen Begriff, denn der ist in seiner Bedeutung meistens klarer definiert, wie zum Beispiel bei »smarten« Zeigern – die Übersetzung mit »schlau« oder Ähnlichem trifft es nicht so ganz.

Für viele Dinge gibt es aber inzwischen auch fest verankerte deutsche Vokabeln, zum Beispiel das *Muster*, der inzwischen geläufige Begriff für »Pattern«. Auch den »Zeiger« bevorzuge ich gegenüber »Pointer«.

Für manches muss man jedoch Begriffe festlegen. Besonderes im Umfeld von C++-eigenen Bezeichnungen bemühe ich mich, für dieses Buch bei einem Begriff zu bleiben, wenn mehrere Begriffe für eine Sache üblich sind. Die vielleicht präziseren Übersetzungen oder seltsame deutsch-englische Mixturen, die ich einsetze, sind:

- *Destruktor* bleibt Destruktor.
- *Kopierkonstruktor* statt Copy-Konstruktor, Copy-Constructor, Copy-C'tor `T(const T&)`
- *Verschiebekonstruktor* statt Move-Konstruktor, Movator oder `T(T&&)`
- *Kopieroperator* oder *Zuweisungsoperator* statt Kopier-Zuweisungsoperator, Copy-Operator, Copy-Assign-Operator oder `T::operator=(const T&)`
- *Verschiebeoperator* statt Verschiebe-Zuweisungsoperator, Verschiebe-Operator, Move-Operator, Move-Assign-Operator oder `T::operator=(T&&)`
- *Concept* benutze ich, wenn ich mich auf die C++20-Sprachfeatures rund um `concept` und `requires` beziehe.

Ansonsten bemühe ich mich vor allem um eine eindeutige Sprache. Sollte mir die Gratwanderung der Eindeutigkeit und Ästhetik mal nicht gelingen, bitte ich das zu entschuldigen.

Kapitel 6
Höhere Datentypen

Kapiteltelegramm

- **`string`**
  Basiszeichenkettentyp in C++
- **`wstring`, `u16string`, `u32string`, `u8string`**
  Stringtypen für internationale Zeichenketten
- **`ostream` und `ofstream`**
  Allgemeiner Ausgabedatenstrom und der Ausgabedatenstrom für eine Datei
- **`istream` und `ifstream`**
  Allgemeiner Eingabedatenstrom und der Eingabedatenstrom für eine Datei
- **`cout`, `cin`, `cerr` und `clog`**
  Vordefinierte Standarddatenströme für die Ein- und Ausgabe
- **`operator<<` und `operator>>`**
  Operatoren für die Ein- und Ausgabe in und aus Datenströmen
- **Manipulator**
  Konstrukt, um das Ein- und Ausgabeformat eines Streams zu verändern
- **Container**
  Ein Container ist ein abstrakter Datentyp, der als Behälter für Daten dient. Die gespeicherten Elemente haben alle den gleichen Typ, den Sie beim Erzeugen des Containers festlegen. Die Schnittstellen der verschiedenen Container sind einander sehr ähnlich.
- **Sequenzcontainer**
  Vor allem `vector` ist ein Allround-Container und erlaubt Zugriff per Zahlenindex.
- **Algorithmus**
  In C++ ist damit meist eine Funktion gemeint, die auf Elementen in Containern arbeitet. Normalerweise wird nicht nur eine Art Container unterstützt.
- **Zeiger**
  Eine Art Referenz auf Werte, die den Zustand »keine Referenz« annehmen können

- **C-Array**

  Eine besondere Form eines Containers, der mit C gut zusammenarbeitet, meist dargestellt durch einen Zeiger und hoffentlich noch das Ende oder die Länge des Arrays.

Zunächst geht es um zwei Gruppen von Datentypen, die nicht in der Sprache selbst eingebaut, sondern Teil der Standardbibliothek sind. Sie benötigen also einen oder mehrere `#include` in Ihrem Quelltext.

Wir beginnen mit Strings und Streams:

- **Zeichenketten**

  In `std::string` speichern Sie Folgen von Zeichen, die Sie manipulieren, einlesen und ausgeben können.

- **Streams**

  Die Standardein- und -ausgabestreams `std::cin` und `std::cout` kennen Sie schon. Diese haben die Typen `std::istream` und `std::ostream`. Verwandte Typen können Sie auch für die Ein- und Ausgabe von Dateien nutzen – und für vieles mehr.

6.1 Der Zeichenkettentyp »string«

Zeichenketten speichern und manipulieren Sie am besten (oder zumindest normalerweise) in einem `std::string`. Dieser befindet sich im Header `<string>` der Standardbibliothek. Achten Sie darauf, dass Sie diesen Header also hinzufügen, bevor Sie `string` verwenden.

»string« und »char*«

In C- und vielen C++-Programmen werden Zeichenketten in `char*` oder `char[]` und deren `const`-Varianten gespeichert. Das ist im Allgemeinen auch in Ordnung, doch gehe ich an dieser Stelle zunächst auf die viel C++-artigere Variante `std::string` ein, der Sie in vielen Fällen den Vorzug vor Zeichenketten geben sollten.

Lassen Sie uns Listing 4.28 ein wenig verändern und die Behandlung von Zeichenketten hinzufügen:

```
// https://godbolt.org/z/vMvs47nfs
#include <iostream>          // cin, cout für Eingabe und Ausgabe
#include <string>            // Sie benötigen diesen Header der Standardbibliothek

void eingabe(
    std::string &name,       // als Parameter
```

```
    unsigned &gebJahr)
{
    /* Eingaben noch ohne gute Fehlerbehandlung... */
    std::cout << "Name: ";
    std::getline(std::cin, name); // getline liest in einen String ein
    if(name.length() == 0) {      // length ist eine Methode von string
        std::cout << "Sie haben einen leeren Namen eingegeben.\n";
        exit(1);
    }
    std::cout << "Geb.-Jahr: ";
    std::cin >> gebJahr;
}
int main() {
    /* Daten */
    std::string name;             // definiert und initialisiert eine string-Variable
    unsigned gebJahr = 0;
    /* Eingabe */
    eingabe(name, gebJahr);
    /* Berechnungen */
    // ...
}
```

Listing 6.1 Einige Verwendungsmöglichkeiten von Strings

Mit Variablen vom Typ `string` können Sie allerlei machen. Es stehen Ihnen Funktionen zur Verfügung, die als Parameter einen `string` nehmen, wie zum Beispiel `std::getline` bei `std::getline(std::cin, name)`. Außerdem hat jede `string`-Variable *Methoden*, die Sie mit dem Punkt `.` aufrufen, wie `name.length()` in dem `if`. Bei Methoden handelt es sich um spezielle Funktionen, die immer auf einer der Variablen arbeiten, mit der sie per Punktnotation aufgerufen wurden.

6.1.1 Initialisierung

Sie sehen am Anfang von `main()`, wie Sie einen `string` definieren. Weil es sich nicht um einen eingebauten Typ handelt, wird dieser auch initialisiert, obwohl kein `=` oder Ähnliches angegeben wurde. In diesem Fall sind die folgenden Varianten gleichbedeutend:

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

Hier sind aber nur die ersten beiden Varianten komplett identisch. Als komplexer Datentyp hat `string` einen Konstruktor – eine spezielle Funktion, die extra dem Zweck dient, eine neue Variable zu initialisieren. Wenn Sie keine Initialisierung angeben, wird der Konstruktor verwendet, der keine Parameter hat. Konstruieren können Sie wahlweise seit C++11 auch mit `{…}`, sodass `{}` den Aufruf des Konstruktors ohne Parameter explizit macht. Mehr dazu finden Sie in Kapitel 12, »Von der Struktur zur Klasse«.

Im Fall von `string` initialisiert dieser Konstruktor die Variable mit dem leeren String, also `""`. Daher bewirkt der Aufruf des Konstruktors mit dem leeren String als Parameter `std::string name{""}` effektiv das Gleiche.

Bei der Initialisierung bedeutet das Gleichheitszeichen `=` nicht »Zuweisung« – stattdessen wird der Konstruktor mit einem Parameter aufgerufen. `std::string name = ""` wirkt also wie `std::string name{""}`.

Vor C++11 standen Ihnen für die Initialisierung von `string` die geschweiften Klammern nicht zur Verfügung. Die Schreibweise im älteren Standard war `std::string name("")`.

Achtung bei der Initialisierung mit runden Klammern ohne Parameter

Sie können *nicht* `std::string name();` zur Initialisierung ohne Parameter verwenden. Benutzen Sie in diesem Fall *immer* die leeren geschweiften Klammern, also `std::string name{};` – bzw. vor C++11 ohne Klammern `std::string name;`.

Die leeren runden Klammern `()` haben hier dummerweise eine Doppelbedeutung als leere Typliste für eine Funktionsdeklaration: Die Funktion oben deklariert also eine Funktion `name` ohne Parameter mit dem Rückgabetyp `string`. Der Compiler akzeptiert dies leider zunächst, meckert aber bei der Verwendung von `name`.

Wenn Sie C++11 oder höher zur Verfügung haben, sollten Sie sich angewöhnen, Variablen immer mit den geschweiften Klammern oder dem Gleichheitszeichen `=` zu initialisieren.

Es gibt noch eine weitere Variante, die den gleichen Effekt hat:

```
std::string name = {""};
```

Verwenden Sie diese Schreibweise, wenn Sie die Initialisierung mit einer *Initialisierungsliste* vornehmen wollen – in diesem Fall ist das eine Liste mit nur einem Element, nämlich dem Leerstring `""`. Da das `=` bei der Initialisierung optional ist, entspricht diese Schreibweise so gut wie immer `std::string name{""}`. Damit Sie die Übersicht behalten, empfehle ich aber, diese Schreibweise nur speziell für die Initialisierungsliste zu verwenden.

6.1.2 Funktionen und Methoden

Sie haben gesehen, dass Sie neue `string`-Variablen zum Beispiel so erzeugen können:

- `std::string name;` oder `std::string name{};` erzeugt einen leeren `string`.
- `std::string name = "Wert";` oder `std::string name{"Wert"};` erzeugt einen `string` aus einem Literal.

Statt `"Wert"` können Sie auch einen anderen `string` verwenden, jenen also *kopieren*. In der Dokumentation zu `string` finden Sie weitere Möglichkeiten zur Initialisierung und noch mehr Funktionen und Methoden für `string`. Tabelle 6.1 und Tabelle 6.2 zeigen Ihnen die wichtigsten auf einen Blick.

Funktion	Beschreibung
`+`	Aneinanderfügen zweier `string`s zu einem neuen
`<<`	Ausgabe eines `string`s
`>>`	Lesen eines `string`s bis zum nächsten Whitespace
`getline`	Lesen in einen `string` bis zum nächsten Newline

Tabelle 6.1 Eine Auswahl an »string«-Funktionen

Methode	Beschreibung
`length`	Länge des Inhalts
`at`	(Sicheres) Holen eines einzelnen Zeichens
`[]`	Holen eines einzelnen Zeichens ohne Prüfung
`find`	Suchen innerhalb des `string`
`find_first_of`	Suche erstes Zeichen aus einer Menge.
`substr`	Erzeuge neuen `string` aus einem Bereich.
`compare`	Vergleiche mit anderem `string`.
`clear`	Zurücksetzen auf den leeren `string`
`append`	Anhängen einer Zeichenfolge
`+=`	Alternative Schreibweise für `append`
`insert`	Einfügen in den `string`

Tabelle 6.2 Eine Auswahl an »string«-Methoden

Methode	Beschreibung
`erase`	Löschen eines Teils des `strings`
`replace`	Ersetzen eines Teils durch eine andere Zeichenfolge
`starts_with`	Prüfe, ob der der Anfang passt, seit C++20.

Tabelle 6.2 Eine Auswahl an »string«-Methoden (Forts.)

6.1.3 Andere Stringtypen

Der gerade vorgestellte `std::string` ist ein Container für einzelne Zeichen vom Typ `char`. Ein `char` hält (meistens) acht Bit und kann somit nur 256 Zustände unterscheiden. Wenn Sie nun jedoch zum Beispiel arabische oder chinesische Schriftzeichen oder vielleicht sogar tolkiensche Zwergenrunen speichern möchten, dann Sie nur die Wahl zwischen diesen Möglichkeiten:

- Sie definieren die Bedeutung der `char`-Werte von 0 bis 255 um und interpretieren sie beim Lesen als Zeichen in der fremden Sprache. Dies nennt man dann eine *Codierung*. Sie müssen so immer die Zusatzinformation der *Codepage* (auch des *Encodings*) mitschleppen – oder wissen.
- Sie betrachten bestimmte Zeichen als besondere Zeichen (*Escapezeichen*), die markieren, dass eine Sequenz von fremden Zeichen folgt. Einfache Wörter (deutsche oder englische) schreiben sich dann 1 zu 1 als `char`, aber die meisten fremden Zeichen bestehen aus zwei oder mehr `char`-Einheiten. Dies ist zum Beispiel in der Codierung »UTF-8« der Fall. Dort kann ein einzelnes Zeichen (»Codepoint«) bis zu sechs `char` lang sein.
- Sie Verwenden einen anderen Elementtyp als `char`, der mehr unterschiedliche Zustände annehmen kann. Dies ist mit `std::wstring` der Fall.

Der Typ `std::wstring` besteht aus Elementen vom Typ `wchar_t`. Dieser ist 16 Bit oder 32 Bit breit und sollte in den allermeisten Fällen für fremde Zeichen ausreichen. (Für *alle* Unicode-Zeichen reichen auch 16 Bit nicht aus, sodass Zeichenfolgen dann zum Beispiel mit UTF-16 codiert werden können.)

Weil aber das Hantieren mit einem Elementtyp, der auf unterschiedlichen Systemen unterschiedlich groß ist, so manchen Programmcode verkompliziert, gibt es auch `std::u16string` und `std::u32string` aus Elementen der Typen `char16_t` und `char32_t`. Mit C++20 kommt noch `std::u8string` auf Basis von `char8_t` hinzu.

Sie werden wahrscheinlich erst dann mit diesen Typen zu tun haben, wenn Sie internationale Programme schreiben. Prinzipiell ist der Umgang mit diesen Stringtypen nicht anders als mit `std::string`, denn sie sind alle nur Synonyme einer Templateklasse `std::basic_string<>`. Setzen Sie zwischen die spitzen Klammern die Element-

typen `char`, `wchar_t`, `char16_t`, `char32_t` oder `char8_t` ein, erhalten Sie die entsprechenden Stringtypen.

Das heißt, alle Methoden, die `std::string` hat, haben die anderen auch. Die freien Funktionen (wie `std::getline()`) wurden ebenfalls so geschrieben, dass sie auf allen Stringtypen arbeiten können.

In der Praxis ist es jedoch oft nicht so leicht, mit etwas anderem als `std::string` umzugehen. Bei den Schnittstellen zur Außenwelt müssen Sie immer sicherstellen, dass die richtige Interpretation ankommt: Erwartet die Ausgabekonsole eine bestimmte Codierung? Was für Strings sind in Dialogboxen? Datenströme aus dem Internet sind meist `char`-basiert, wie wird daraus etwas anderes?

Dies ist ein sehr weites Feld. Lesen Sie sich in den systemspezifischen Bereich ein, den Sie benötigen, oder lassen Sie sich den Auszug, den Sie brauchen, von einem Kollegen erklären.

6.1.4 Nur zur Ansicht: »string_view«

Strings können lang sein. Sehr lang. Und modernes C++ propagiert, dass Sie mit Werten statt Zeigern oder Referenzen programmieren. Das ist bei potenziell riesigen Objekten eine Gretchenfrage – es ist nicht weise bei langen Strings rigoros Werte einzusetzen. Was also tun? Die in C++17 neue `string_view` ist die Lösung dazu.

Eine `string_view` (und die entsprechenden anderen `wstring_view` etc.) ist ein Nur-Lese-Objekt in einen `string`. Die Klasse bietet dieselben Funktionen und Methoden an, solange diese nur lesend sind. So können Sie iterieren mit `begin()` und `end()`, finden mit `find()`, vergleichen mit `compare()` oder `==`, den Anfang überprüfen mit `starts_with()` und vieles mehr – und in Algorithmen verwenden.

Bestimmte Modifikationen können Sie aber doch durchführen. Sie können vom Anfang und vom Ende etwas wegschneiden. Mit `remove_prefix()` und `remove_suffix()` können Sie die `string_view` vorne oder hinten um eine Anzahl Zeichen verkürzen.

Eine `string_view` benötigt extrem wenig Ressourcen, nämlich genau einen Zeiger in einen bestehenden String und eine Länge. Vor allem wird absolut kein Heap benötigt. Daher eignet sich eine `string_view` besonders als Funktionsparameter. Hier ist es praktisch, dass sich eine `string_view` mittels einer automatischen Typumwandlung aus einen `string` leicht erzeugen lässt:

```
// https://godbolt/.org/z/bdaE5nf5T
#include <iostream>
#include <string>
#include <string_view>
void zeige_mitte(std::string_view msg) {      // string_view ist ein guter Parameter
   auto mitte = msg.substr(2, msg.size()-4); // substr liefert string_view zurück
```

```
   std::cout << mitte << "\n";
}
int main() {
    using namespace std::literals;
    const std::string aaa = "##Etwas Text##"s;
    zeige_mitte(aaa);                          // Umwandlung in string_view
    auto bbb = "++Mehr Text++"sv;              // string_view als Literal
    zeige_mitte(bbb);
}
```

Listing 6.2 Eine string_view lässt sich aus einen string leicht erzeugen.

Bei `zeige_mitte(aaa)` wandelt der Compiler den `string` namens `aaa` automatisch in eine `string_view` um. Mit weniger Ressourcen kann man einen `string` beinahe nicht als Parameter übergeben. Wie Sie bei `msg.substr` sehen, hat auch `string_view` die von `string` bekannte Methode. Die Rückgabe `mitte` ist eine neue `string_view`.

Bemerkenswert ist, dass Sie mit dem Suffix `""sv` direkt eine `string_view` als Literal erzeugen können. `bbb` zeigt somit in den Quelltext (bzw. in den statischen Teil des kompilierten Programms) und nimmt so gut wie keinen eigenen Speicher weg.

Auch als Rückgabewert können Sie `string_view` manchmal verwenden. Hier müssen Sie aber genau wie bei Zeigern und Referenzen aufpassen, dass der originale `string`, auf dem die `string_view` basiert, auch noch existiert.

6.2 Streams

Sie haben Streams in Beispielen schon oft gesehen, aber selten trat ihr Typ in Erscheinung, weil wir die vordefinierten *Streams* (engl. für *Datenströme*) der Standardbibliothek verwendet haben:

- `std::cout` vom Typ `std::ostream` dient der *Standardausgabe*, meist auf die Konsole, also den Bildschirm. Oder Sie können die Ausgabe in eine Datei umleiten.
- `std::cin` vom Typ `std::istream` dient der *Standardeingabe*, üblicherweise von der Tastatur oder einer umgeleiteten Datei.
- `std::cerr` und `std::clog` sind ebenfalls Ausgabedatenströme vom Typ `std::ostream` und landen meist auf dem Bildschirm, doch Sie können sie getrennt von `std::cout` in eine Datei umleiten.

Für Ein- und Ausgaben des normalen Programmflusses verwenden Sie normalerweise `std::cin` und `std::cout`. Jede Ausgabe kostet Zeit, und ein Aufruf hat »Overhead«, daher ist `std::cout` gepuffert – Ausgaben erscheinen manchmal nicht sofort, sondern werden erst gesammelt.

Anders `std::cerr`, dessen Ausgabe soll sofort erscheinen. Verwenden Sie diesen Stream für Fehlerausgaben. Wenn `std::cout` in eine Datei umgelenkt wurde, dann erscheinen diese Ausgaben trotzdem auf der Konsole, wie in Listing 6.3 gezeigt.

```
// https://godbolt/.org/z/raYaeY357
// Rufen Sie dieses Programm zum Beispiel mit 'prog.exe > datei.txt' auf.
#include <iostream>  // cout, cerr
int main() {
    std::cout << "Ausgabe nach cout\n";       // wird nach 'datei.txt' ausgegeben
    std::cerr << "Fehlermeldung!\n";          // erscheint trotzdem auf der Konsole
    std::cout << "Wieder normale Ausgabe\n"; // wieder in die Datei
}
```

Listing 6.3 Der Fehlerstream ist von der Standardausgabe getrennt.

6.2.1 Eingabe- und Ausgabeoperatoren

Wie Sie schon häufig gesehen haben, können Sie mit dem Operator << Daten nach `cout` ausgeben. Alle eingebauten Datentypen lassen sich so ausgeben, Sie müssen sich nicht um das Format kümmern – können es aber tun, wie Sie gleich bei den *Manipulatoren* sehen werden. Das ist erwähnenswert, weil anders als in der C-Funktion `printf()` der Compiler für Sie das korrekte Format wählt – nämlich die richtige Überladung des globalen `operator<<`. Sie können für eigene Datentypen auch Überladungen hinzufügen, siehe Abschnitt 12.11, »Eigene Datentypen verwenden«.

Sie können mehrere <<-Aufrufe hintereinanderketten, Sie müssen nur als ersten den Stream nennen, in den ausgegeben werden soll: Jeder <<-Aufruf gibt den Stream als Ergebnis zurück, sodass dieser wieder eine Ausgabe empfangen kann.

Für die Eingabe per >> gilt das Gleiche – gelesen wird für die eingebauten Datentypen immer bis zum nächsten Whitespace (zum Beispiel Leerzeichen oder Zeilenvorschub).

Erweitern wir Listing 6.1 um ein paar zusätzliche Ein- und Ausgaben:

```
// https://godbolt/.org/z/Gbe7aM4o4
#include <iostream>                    // cin, cout für Eingabe und Ausgabe
#include <string>
#include <array>
using std::cin; using std::cout;       // Abkürzungen cin und cout
void eingabe(
    std::string &name,
    unsigned &gebTag,
    unsigned &gebMonat,
    unsigned &gebJahr,
```

```
    long long &steuernummer,
    std::array<int,12> &monatseinkommen) // array ist ein Container
{
    /* Eingaben noch ohne gute Fehlerbehandlung... */
    cout << "Name: ";
    std::getline(cin, name);    // getline nimmt Eingabestrom und String
    if(name.length() == 0) {
        cout << "Sie haben einen leeren Namen eingegeben.\n";
        exit(1);
    }
    cout << "Geb.-Tag: "; cin >> gebTag;
    cout << "Geb.-Monat: "; cin >> gebMonat;
    cout << "Geb.-Jahr: "; cin >> gebJahr;
    cout << "Steuernummer: "; cin >> steuernummer;
    for(int m=0; m<12; ++m) {
        cout << "Einkommen Monat " << m+1 << ": "; // mehrere Ausgaben
        cin >> monatseinkommen[m];                 // Einlesen mit Operator
    }
    cout << std::endl;
}
int main() {
    std::string name{};
    unsigned tag = 0;
    unsigned monat = 0;
    unsigned jahr = 0;
    long long stNr = 0;
    std::array<int,12> einkommen{};
    eingabe(name, tag, monat, jahr, stNr, einkommen);
    // ... Berechnungen ...
}
```

Listing 6.4 Eine Funktion mit Ein- und Ausgabe

Hier sehen Sie eine Menge Ausgaben per `std::cout << ...` Egal, ob es sich um eine Zeichenkette wie `"Geb.-Tag: "` oder um eine Zahl wie `m+1` handelt, Sie verwenden immer einfach `<<`. Bei `cout << "Einkommen Monat "...` sehen Sie auch, wie mehrere Ausgaben hintereinandergekettet werden.

Die Eingabe von der Tastatur geht ebenso leicht: Der `>>`-Operator, gefolgt von der Variablen, in die Sie hineinlesen wollen, lässt die Benutzer einen Wert eingeben. Hier ist jedoch etwas Vorsicht gefragt: Eingaben sollten jeweils mit einem Druck auf die ↵-Taste abgeschlossen werden. Für die eingebauten Datentypen ist beim nächsten Whitespace – wie zum Beispiel Leertaste oder ↵ – die Eingabe für die Variable zu

Ende. Kommt ein Leerzeichen in der Benutzereingabe vor, wird alles dahinter für die nächste Ausgabeoperation per >> zurückgehalten – und alles kommt durcheinander.

6.2.2 »getline«

Das ist auch der Grund dafür, warum ich bei `std::getline(cin, name);` für `name` nicht >> verwende, sondern `getline()`. Diese Funktion liest nicht nur bis zum nächsten Whitespace, sondern bis zum nächsten Zeilenende – und erlaubt somit auch Namen mit Leerzeichen.

6.2.3 Dateien für die Ein- und Ausgabe

Wenn Sie ein Programm aufrufen, können Sie `cout` mit `> datei` in eine Datei »umlenken«. Das folgende Programm wandelt die Argumente in Großbuchstaben um und gibt sie aus:

```
//https://godbolt.org/z/a6qqbWKMs
#include <cstdlib>
#include <iostream>

int main(int argc, char* argv[]) {
    for(int i=1; i<argc; ++i) {                // Start bei 1
        for(char* p=argv[i]; *p!='\0'; ++p) {
            char c = toupper(*p);
            std::cout << c;
        }
        std::cout << ' ';
    }
    std::cout << '\n';
}
```

Die Schleife mit `i` läuft über alle Argumente, die Sie dem Programm mitgeben, die Schleife mit `p` über alle Zeichen der einzelnen Argumente. Jedes Argument ist bei `\0` zu Ende. Die Funktion `toupper()` wandelt einen `char` in einen Großbuchstaben um. So rufen Sie das Programm auf (das `$`-Prompt geben Sie nicht mit ein):

```
$ .\caps.exe Hallo Text
```

Das schreibt es auf den Bildschirm:

```
HALLO TEXT
```

Diese Ausgabe können Sie umlenken:

```
$ .\caps.exe Hallo Text > ausgabe.txt
```

Statt dass Sie die Ausgabe auf dem Bildschirm sehen, wird sie in `ausgabe.txt` geschrieben. Unter Unix ist dieses Vorgehen mehr als üblich, für Windows-Benutzer ist es jedoch etwas gewöhnungsbedürftig. Aber da Sie ja nun C++ programmieren und die Bedienung eines Texteditors beherrschen, sollte Ihnen der Umgang mit der Kommandozeile keine Schwierigkeiten bereiten.

Die Datei `ausgabe.txt` können Sie in einem beliebigen Texteditor – auch in der C++-Entwicklungsumgebung – einsehen. Unter Unix schreiben Sie statt `.\programm.exe` natürlich `./programm`.

Wollen Sie aus einer Datei lesen, können Sie `< datei` verwenden:

```
$ .\programm.exe < eingabe.txt
```

Wollen Sie as aber nicht den Aufrufenden überlassen, die Datei für das Lesen oder Schreiben zu wählen, oder benötigen Sie gar mehr als eine Ein- und Ausgabe, dann können Sie sich im Programm neue Datenströme erzeugen.

Erzeugen Sie eine neue Datei mit dem Typ `std::ofstream` und geben Sie den gewünschten Dateinamen an. Statt `<iostream>` inkludieren Sie `<fstream>`, wo die Dateistreams definiert sind:

```
//https://godbolt.org/z/McqfcWb8j
#include <fstream>
int main(int argc, char* argv[]) {
    std::ofstream meineAusgabe{"output1.txt"};
    meineAusgabe << "Zeile 1\n";
    meineAusgabe << "Zeile 2\n";
}
```

Die Datei wird geschlossen, sobald die Variable `meineAusgabe` nicht mehr gültig ist, also zum Beispiel, wenn ihr Block verlassen wird.

Das Gleiche gilt für das Lesen bestehender Dateien, nur verwenden Sie dazu `ifstream`:

```
//https://godbolt.org/z/P9z8n8v16
#include <fstream>
int main(int argc, char* argv[]) {
    int wert = 0;
    std::ifstream meineEingabe{"input1.txt"};
    meineEingabe >> wert;
}
```

Sollte die Datei nicht existieren, wird das nicht funktionieren. Sie können prüfen, ob das Öffnen der Datei einen Fehler verursacht hat, indem Sie den Nicht-Operator `!` auf den Stream anwenden:

```
//https://godbolt.org/z/ssW3eKEGq
#include <iostream> // cerr
#include <fstream>
int main(int argc, char* argv[]) {
    int wert;
    std::ifstream meineEingabe{"input1.txt"};
    if(!meineEingabe) {
        std::cerr << "Fehler beim Öffnen der Datei!\n";
    } else {
        meineEingabe >> wert;
    }
}
```

Listing 6.5 Verwenden Sie den !-Operator, um den Zustand des Streams zu prüfen.

6.2.4 Manipulatoren

Sollte Ihnen das Format der Ausgabe der Standarddatentypen nicht gefallen, können Sie es vielfältig beeinflussen. Im Header <iomanip> finden Sie allerlei *Manipulatoren*, die Sie einfach auf den Stream anwenden, um das Verhalten umzuschalten.

Dies ist gerade bei Fließkommazahlen interessant, um die Anzahl der ausgegebenen Nachkommastellen zu beeinflussen. Listing 6.6 gibt ein kurzes Beispiel:

```
//https://godbolt.org/z/EcnWz3r3j
#include <iostream>
#include <iomanip>                    // fixed, setprecision
#include <format>                     // C++20
using std::cout; using std::format;   // Abkürzung cout, format
int main() {
    cout << std::fixed                // Punktschreibweise, nicht wissenschaftlich
         << std::setprecision(15);    // 15 Nachkommastellen
    cout << 0.5 << "\n";              // Ausgabe: 0.500000000000000*
    cout << std::setprecision(5);     // 5 Nachkommastellen
    cout << 0.25 << "\n";             // Ausgabe: 0.25000
    cout << format("{:0.4f}", 0.75) << "\n"; // (C++20)  Ausgabe:  0.7500
    return 0;
}
```

Listing 6.6 Verwenden Sie Streammanipulatoren aus <iomanip>, um das Format der Ausgabe zu beeinflussen.

Für die komplette Liste der Manipulatoren schauen Sie sich Tabelle 6.3 an. Sie finden eine ausführlichere Beschreibung der Manipulatoren in Abschnitt 27.5, »Streams manipulieren und formatieren«. Seit C++20 haben Sie auch die Möglichkeit, die Forma-

tierung mit `std::format` zu erledigen. Die Formatierungsangaben sind besser lesbar und von anderen Sprachen wie Python (oder so ähnlich auch C) bekannt.

Bezeichner	Beschreibung
Header <ios>	
`[no]boolalpha`	textuelles oder numerisches `bool`
`[no]showbase`	Zahlen mit oder ohne Präfix
`[no]showpoint`	Fließkomma immer mit Punkt
`[no]showpos`	positive Zahlen mit `+` anzeigen
`[no]skipws`	führende Whitespaces beim Lesen überspringen
`[no]uppercase`	Großbuchstaben für manche Ausgaben
`[no]unitbuf`	Flush nach jeder Ausgabe?
`left right internal`	Wo sollen Füllzeichen hin?
`dec hex oct`	Zahlen in anderer Basis ausgeben
`fixed defaultfloat scientific hexfloat`	Ausgabeformat von Fließkommazahlen
Header <istream>	
`ws`	alle Whitespaces konsumieren
Header <ostream>	
`ends`	gibt `\0` aus
`flush`	gepufferte Ausgabe sofort ausgeben
`endl`	gibt `\n` aus und flusht
Header <iomanip>	
`[re]setiosflags`	angegebene I/O-Flags (zurück-)setzen
`setbase`	Ganzzahlen in anderer Basis ausgeben
`setfill`	Füllzeichen verändern
`setprecision`	Ausgabegenauigkeit von Fließkommas
`setw`	Breite der Ausgabe setzen

Tabelle 6.3 Die Streammanipulatoren

Bezeichner	Beschreibung
`put/get_money/time`	I/O besonderer Zahlenformate
`quoted`	Anführungszeichen bei Ein- und Ausgabe

Tabelle 6.3 Die Streammanipulatoren (Forts.)

6.2.5 Der Manipulator »endl«

Nicht alle Manipulatoren verändern tatsächlich das Lese- oder Schreibformat. Die wichtigste Ausnahme ist `std::endl`. Er wirkt wie ein Zeilenendezeichen `\n`, hat aber den zusätzlichen Effekt, dass er den Schreibpuffer leert – und zwar, indem er noch anstehende Schreibaufgaben ausführt. Dies kostet Zeit – schließlich wurde Pufferung ja extra zur Beschleunigung erfunden. Deshalb sollten Sie, wenn es geht, besser `\n` zum Zeilenvorschub verwenden.

Nur wenn Sie sicher sein wollen, dass eine Bildschirmausgabe zu einem bestimmten Zeitpunkt bei Ihren Benutzern auch ankommt, sollten Sie stattdessen `std::endl` verwenden.

Wollen Sie nur den zurückgehaltenen Puffer ausgeben und keinen zusätzlichen Zeilenvorschub, dann benutzen Sie den Manipulator `flush`, also zum Beispiel `cout << flush`.

Bevor eine Datei geschlossen wird, also das Programm endet oder die Streamvariable ungültig wird, benötigen Sie kein zusätzliches `flush` oder `endl`. Das Schließen der Datei leert den Puffer automatisch. Nur wenn Ihr Programm abstürzt, könnten ungeschriebene Reste im Puffer verbleiben.

6.3 Behälter und Zeiger

Container sind ein wichtiger Bestandteil der Standardbibliothek. Weil sie eine zentrale Rolle spielen und auch viel Platz darin einnehmen, gehe ich im Detail später in Kapitel 24, »Container«, auf sie ein. Hier gebe ich Ihnen einen kurzen Überblick, damit Sie wissen, welche Räder Sie nicht neu erfinden müssen.

6.3.1 Container

Sie haben bisher Typen kennengelernt, die ein einzelnes Datum (im Sinne von *Daten*) halten können. Wenn Sie zum Beispiel zwei `int`-Werte speichern wollten, dann haben Sie dafür zwei Variablen (wie `int x, y;`) gebraucht. Mit `array<int>` hatten Sie schon einen Vorgeschmack auf einen der Container, und ich erkläre Ihnen dazu die ersten Hintergründe.

Was tun Sie, wenn Sie richtig viele Werte speichern wollen? Oder Sie wollen vielleicht über alle Werte in einer Schleife iterieren? Oder Sie wissen zuvor vielleicht nicht, wie viele Werte es werden? Dafür gibt es in der C++-Standardbibliothek die *Container* (engl. für *Behälter*).

Merken Sie sich, dass Container in der C++-Standardbibliothek ein Konzept sind. Die jeweiligen Schnittstellen aller Container sind gleich, und Sie können das, was Sie über einen Container wissen, auf alle anderen übertragen – das Einhalten einiger Regeln vorausgesetzt. Es gibt Ausnahmen, und deren nicht wenige, aber sobald Sie das Konzept verstanden haben, werden Sie alle Container meistern.

Es gibt viele Behälter in der Standardbibliothek, und je nach Zweck der Anwendung ist der eine oder der andere sinnvoll. Ich gehe in diesem Kapitel auf zwei ein, weil sie einerseits fundamental und andererseits zueinander sehr unterschiedlich sind.

6.3.2 Parametrisierte Typen

Alle Container haben gemeinsam, dass sie *parametrisierte Typen* sind. Sie bestehen aus einem Haupttyp – dem eigentlichen Container – und dem Typ oder den Typen, die in den Container hineingesteckt werden.

Dem Haupttyp folgen die Parameter in spitzen Klammern, zum Beispiel `vector<int>`, `map<string,Car>` oder `array<double,10>`. Es ist höchst wichtig, dass Sie das gesamte Konstrukt als *einen* Typ betrachten. Ein `vector<int>` ist etwas anderes als ein `vector<long>`, genauso wie ein `int` etwas anderes als ein `long` ist. Für die Überladung von Funktionen und die Auflösung von Operatoren ist das von entscheidender Bedeutung. Sie können zum Beispiel zwei Funktionen wie diese schreiben:

```
long sum(vector<int> arg);
long sum(vector<long> arg);
```

Das geht, weil die Argumenttypen unterschiedlich sind. Sie schreiben damit eine ganz normale Überladung (siehe Abschnitt 7.8).

Das gilt auch für (konstante) Werte als Parameter: `array<int,10>` ist ein anderer Typ als `array<int,11>`.

Als Konsequenz daraus, dass *haupttyp<parameter>* als Ganzes den Typ bildet, muss dieser schon zur Übersetzungszeit feststehen. Das fällt Ihnen besonders bei `array` auf. Vielleicht sind Sie versucht, eines der folgenden Dinge zu tun:

```
void berechneImArray(int n) {
    array<int,n> daten {};  //↗ n muss konstant sein
    // ...
}
```

```
long summiereArray(array<int,n> data) { //↯ wo kommt dieses n her?
    int sum = 0;
    for(int e : data)
        sum += e;
    return sum;
}
```

Beides geht nicht, da `n` jeweils eine erst zur Laufzeit feststehende Variable ist. Sie können höchstens einfache Berechnungen wie `3+4` oder eine `constexpr` verwenden.

Doch genug des Vorausblicks auf `array` als Vehikel für das Konzept der parametrisierten Typen. Ich werde nun besser konkret.

Nennung des Elementtyps eingespart

Beachten Sie, dass Sie sich mit C++17 häufig die Nennung der Typparameter zwischen den spitzen Klammern sparen können. Der Compiler kann mithilfe der Konstruktorargumente den Elementtyp des Containers herausfinden:

```
// https://godbolt.org/z/1r67vGh1M
std::vector vec { 1, 2, 3 };
```

Es handelt sich hierbei immer noch um einen `vector<int>`, nur dass Sie sich etwas Tipparbeit sparen können. Das ist aber nicht immer sinnvoll (oder möglich).

```
std::vector<std::string> elben { "Elrond", "Galadriel" };
using namespace std::literals;
std::vector zwerge { "Oin"s, "Gloin"s };
```

Mit dem Suffix `""s` erzwingen Sie `string`-Konstruktorargumente und erhalten einen `vector<string>` auch für `zwerge`.

6.4 Die einfachen Sequenzcontainer

- **`std::array`**

  Diesem Behälter sagen Sie beim Entstehen, wie viele Elemente er enthalten soll. Er wächst und schrumpft nicht; auf seine Elemente greifen Sie mit einem Index zu oder iterieren über Bereiche.

- **`std::vector`**

  Dieser Allrounder legt seine Elemente direkt hintereinander ab. Sie greifen über einen Zahlindex auf seine Elemente zu oder iterieren über Bereiche. Außerdem wächst er automatisch, wenn Sie Elemente hinzufügen.

Ich bemühe mich, nicht von *Arrays* zu reden, sondern immer entweder `array` für `std::array` oder C-Array für `typ[]` zu schreiben. In anderen Quellen wird Letzteres meist einfach »Array« genannt oder in deutschen Texten auch »Feld«.[1]

6.4.1 »array«

Das `array` verwenden Sie, wenn Sie im Vorfeld wissen, wie viele Elemente Sie benötigen. Wie Sie in der ersten Zeile von `main()` sehen, legen Sie die Anzahl der Elemente bei der Deklaration fest. Sie kann nicht wieder geändert werden.

```
// https://godbolt.org/z/cd1d1W7T1
#include <array>
#include <iostream>
using std::cout; using std::array; using std::string;
int main() {
  array<string,7> wotag = { "Montag", "Dienstag",         // definieren
      "Mittwoch", "Donnerstag", "Freitag", "Samstag", "Sonntag" };
  cout << "Die Woche beginnt mit " << wotag[0] << ".\n"; // Werte lesen
  cout << "Sie endet mit " << wotag.at(6) << ".\n";      // sicheres Werte lesen
  /* nordisch? */
  wotag[5] = "Sonnabend";                                // Werte verändern
}
```

Listing 6.7 In einem »array« speichern Sie eine feste Anzahl Elemente.

Auf die Elemente greifen Sie mit eckigen Klammern bei `wotag[0]` und `wotag[5]` über einen Zahlenindex zu – Sie können sowohl Werte lesen als auch schreiben. Die alternative Methode `at(index)` ist etwas sicherer: Während mit `[]` keine Überprüfung stattfinden muss, ob Sie einen zu großen oder zu kleinen Index verwendet haben, und Ihr Programm möglicherweise abstürzt oder Schlimmeres passiert (es mit falschen Werten weiterläuft), enthält `at()` eine Überprüfung, mit der Sie einen Fehler abfangen können – oder wenigstens vor Programmende eine Fehlermeldung erhalten (siehe Kapitel10, »Fehlerbehandlung«).

Die Anzahl der Elemente bei der Deklaration von `wotag` muss eine Konstante sein. Sie können hier keine Variable verwenden, deren Wert Sie zuvor ermittelt haben. Die `7` ist als Zahlenliteral natürlich konstant. Es ist aber guter Stil, eine solche Zahl als Konstante vorher zu deklarieren:

```
// https://godbolt.org/z/zfe7dM4ex
#include <array>
```

1 Andere deutsche Texte sagen »Feld« zum `vector`. Wegen all dieser Verwirrungen verwende ich in diesem Buch bevorzugt die originalen Begriffe der Sprache C++.

```
#include <iostream>
constexpr size_t MONATE = 12; /* Monate im Jahr */
int main() {
    std::array<unsigned,MONATE> mtage = {      // okay mit einer Konstante
        31,28,31,30,31,30,31,31,30,31,30,31};
    unsigned alter = 0;
    std::cout << "Wie alt sind Sie? "; std::cin >> alter;
    std::array<int,alter> lebensjahre;        // Arraygröße geht nicht per Variable
}
```

Listing 6.8 Die Arraygröße muss konstant sein.

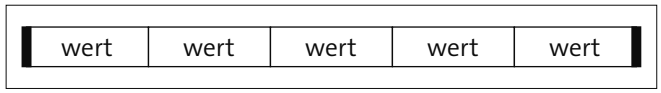

Abbildung 6.1 Ein »array« kann weder wachsen noch schrumpfen. Seine Elemente sind kompakt angeordnet.

Wenn Sie ein `array` als Parameter an eine Funktion übergeben wollen, dann muss dessen Typ mit der Arraydefinition genau übereinstimmen. Denn `array<int,4>` ist ein anderer Typ als `array<int,5>` und `array<short,4>` – auf einen solchen Parametertyp würde es nicht passen. Damit Sie sich nicht ständig wiederholen müssen, sollten Sie daher zuvor dem Arraytyp mit einem `using` (oder `typedef`) einen eigenen Namen geben:

```
// https://godbolt.org/z/a4zxbd99z
#include <array>
#include <algorithm>                    // accumulate
#include <numeric>                      // iota
using Januar = std::array<int,31>;      // Alias für wiederholte Verwendung

void initJanuar(Januar& jan) {          // das genaue Array als Parameter
    std::iota(begin(jan), end(jan), 1); // füllt mit 1, 2, 3 ... 31
}
int sumJanuar(const Januar& jan) {      // das genaue Array als Parameter
    return std::accumulate(begin(jan), end(jan), 0); // Hilfsfunktion für Summe
}
int main() {
    Januar jan;                         // deklariert ein array<int,31>
    initJanuar( jan );
    int sum = sumJanuar( jan );
}
```

Listing 6.9 Wenn Sie eine Arraydefinition mehrfach verwenden müssen, dann verwenden Sie »using«.

Müssen Sie eine feste Zahl von gleichförmigen Elementen speichern, eignet sich ein Array hervorragend. Besonders als Membervariable leistet es gute Dienste, gerne auch für statische Daten, die Sie literal in den Quelltext tippen.

Die besondere Stärke ist, dass die Daten direkt nebeneinander im Speicher liegen, was in vielen Einsatzgebieten nützlich ist – zum Beispiel kann man es besonders schnell als einen Block auf die Festplatte schreiben etc.

Ansonsten sehen Sie hier die praktischen Funktionen `accumulate` und `iota` aus den Headern `<algorithm>` und `<numeric>`, die beide sogenannte »Algorithmen« sind. Das sind Hilfsfunktionen, die Sie auf viele, wenn nicht alle Container anwenden können. Sie werden an anderen Stellen im Buch noch mehr darüber erfahren. In Abschnitt 25.6, »Liste der Algorithmusfunktionen und Range-Adapter«, werden sie alle im Detail aufgelistet.

6.4.2 »vector«

Es ist jedoch unpraktisch, immer eine vorher genau festgelegte Zahl an Elementen im Container haben zu müssen. Das Allroundtalent der Behälter ist zweifellos der `vector`. Er enthält wie ein `array` nur Elemente eines Typs. Auch der Zugriff funktioniert genauso, nämlich über einen Zahlenindex (bei 0 beginnend) oder über einen Iterator.

Die Größe des `vector` ist aber dynamisch: Sie kann bei Bedarf wachsen. So können Sie ihn zum Beispiel leer initialisieren und dann nacheinander so viele Elemente hineintun, wie Ihr Computer Speicher hat:

```
// https://godbolt.org/z/T9jsEoMj4
#include <vector>                    // Sie benötigen diesen Header
int main() {
    std::vector<int> quadrate{};     // leer initialisieren
    for(int idx = 0; idx<100; ++idx) {
        quadrate.push_back(idx*idx); // Anfügen eines Elements
    }
}
```

Mit `push_back()` fügen Sie am Ende des `vectors` ein Element an. Das ist neben dem Indexzugriff eine der nützlichsten Funktionen von `vector`, zum Beispiel `quadrate[idx]`.

In einem `vector` liegen die Elemente direkt hintereinander im Computerspeicher. Das ist CPU-freundlich und macht ihn beim Zugriff von vorne nach hinten enorm schnell. Der `vector` hat aber folgenden Nachteil: Wenn Sie ein Element in der Mitte oder vorne einfügen wollen, werden alle Elemente rechts davon um eine Position zur Seite verschoben – und das ist meist eine zeitraubende Angelegenheit.

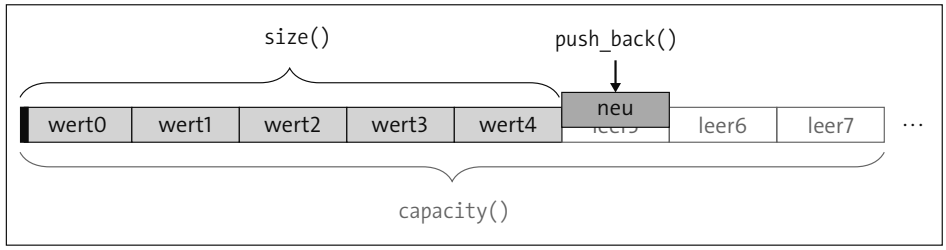

Abbildung 6.2 Schematische Darstellung eines »vectors«.

Bevorzugen Sie das Einfügen von Elementen hinten in einen »vector«

Wann immer Sie können, sollten Sie in einen `vector` Elemente nur hinten einfügen. Die Methoden dafür sind `push_back()` und `emplace_back()`.

Für Operationen auf allen Elementen eines `vector`s stehen Ihnen mehrere Möglichkeiten zur Verfügung. Die bereichsbasierte `for`-Schleife kennen Sie schon:

```
// https://godbolt.org/z/hhhavG5he
#include <vector>
#include <iostream>                        // cout, endl
int main() {
    std::vector quadrate{1,4,9,16,25};   // gefüllt initialisieren
    for(int zahl : quadrate)  // zahl ist ein Quadrat nach dem anderen
        std::cout << zahl << " ";
    std::cout << std::endl;
}
```

Listing 6.10 Die einfachste Iteration benutzt eine bereichsbasierte »for«-Schleife.

Hier werden alle Elemente nacheinander ausgegeben: `1 4 9 16 25`. Sie können alternativ auch per Index auf die Elemente zugreifen. Vergessen Sie nicht, dass die Zählung (fast immer in C++) bei null beginnt. Mit `size()` können Sie prüfen, wie viele Elemente im `vector` sind.

Beachten Sie, dass ich hier `std::vector` statt `std::vector<int>` geschrieben habe, weil der Compiler seit C++17 `<int>` aus den Konstruktorargumenten ermittelt.

```
// https://godbolt.org/z/xcKM7v1Eo
#include <vector>
#include <iostream>                              // cout, endl
int main() {
  std::vector qus{1,4,9,16,25};
  for(unsigned idx=0; idx<qus.size(); ++idx)  // size enthält die Anzahl
    std::cout << qus[idx] << " ";              // [idx] oder at(idx) holt ein Element
```

```
  std::cout << std::endl;
}
```

Listing 6.11 Der Zugriff auf die Elemente per Index

In C++ sollten Sie statt eines Index aber Iteratoren verwenden. Ihr Einsatzgebiet ist breiter, für den Compiler eventuell besser zu handhaben und vor allem: bei allen Containern gleich.

```
// https://godbolt.org/z/1WG8zvhYe
#include <vector>
#include <iostream>                    // cout, endl
int main() {
  std::vector qus{1,4,9,16,25};
  for(auto it = qus.begin(); it!=qus.end(); ++it) // zwischen begin() und end()
    std::cout << *it << " ";           // mit *it kommen Sie vom Iterator zum
Element
  std::cout << std::endl;
}
```

Listing 6.12 Der Einsatz von Iteratoren für eine Schleife

Ebenso wie Container sind *Iteratoren* ein Konzept. Sie arbeiten eng mit Containern zusammen und werden im Detail auch in Kapitel 24, »Container«, erklärt. Bis dahin will ich sie Ihnen auf die kürzestmögliche Art zusammenfassen:

- **`qus.begin()` und `qus.end()`**
  liefern Iteratoren für den Anfang und das Ende des Containers zurück, ähnlich wie `0` und `size()` die Grenzen der Indexe liefert.
- **`auto it = qus.begin()`**
  definiert einen Iterator und initialisiert ihn auf den Containeranfang.
- **`it != qus.end()`**
  prüft auf »bin ich schon am Ende«. Prüfen Sie immer mit »ungleich«, nie mit »kleiner«.
- **`*it`**
  kommt vom Iterator zum Element. Sie »dereferenzieren« den Iterator.

> **Die Standardbibliothek enthält viele Container**
>
> Die weiteren Container und jede Menge Anleitungen für den Umgang mit ihnen finden Sie in Kapitel 24, »Container«.

6.5 Algorithmen

Zusätzlich zu den Fähigkeiten der Container gibt es *Algorithmen*. Die meisten finden Sie im Header `<algorithm>`. Dort gibt es eine lange Liste an Hilfsfunktionen, die Ihnen dabei helfen, spezielle Probleme mit allgemeinen Methoden zu lösen.

Alle diese Algorithmen arbeiten durchgehend auf Iteratoren, weswegen sie auf (beinahe) allen Containern arbeiten. Es ist wichtig zu wissen, dass mit den Methoden der Container deren Mächtigkeit nicht erschöpft ist.

Im folgenden Beispiel wird ein `vector` mit einem Algorithmus gefüllt, um dann Elemente mit einem bestimmten Kriterium zu zählen:

```
// https://godbolt.org/z/4G67rso8T
#include <vector>
#include <algorithm>                          // count_if
#include <numeric>                            // iota
#include <iostream>
bool even(int n) { return n%2==0; }           // Test auf gerade
int main() {
    std::vector<int> data(100);               // 100 x null
    std::iota(data.begin(), data.end(), 0);   // 0, 1, 2, ... 99
    // zählt gerade Zahlen
    std::cout << std::count_if(data.begin(), data.end(), even);
}
```

Listing 6.13 Zählen mit einem Algorithmus

6.6 Zeiger und C-Arrays

Bei der Besprechung der eingebauten Datentypen habe ich zwei sehr wichtige Typen bisher ausgelassen: die Zeiger und die C-Arrays. Diese beiden sind eng miteinander verwandt, welhalb ich sie auch gemeinsam behandle. Weshalb das so spät passiert, hat zwei Gründe:

- Sie sind ein eigenes Konzept und bringen einen riesigen Bereich von Dingen mit, die Sie verstehen müssen, um sie effektiv nutzen zu können.
- Für die Aufgaben, die sie lösen, sind sie in modernem C++ nur die zweitbeste Möglichkeit. Spätestens seit C++11 gibt es meistens eine bessere Alternative in der Sprache oder der Standardbibliothek.

Daher gebe ich Ihnen nur eine kurze Einführung in Zeiger und C-Arrays. Sie lernen mehr in Kapitel 20, »Zeiger«– und erfahren gleichzeitig dann auch noch mehr über die Alternativen.

6.6.1 Zeigertypen

Ein *Zeiger* (engl. *Pointer*) ist die C-Variante einer Referenz, die historisch bedingt auch in C++ noch häufig genug verwendet wird. Es ist eine Indirektion auf den wirklichen Wert. Daher kann es jeden Typ auch als Zeiger geben – sogar Zeiger selbst, also als einen Zeiger auf einen Zeiger. Ein Zeiger repräsentiert eine Adresse im Speicher, und der Computer kann damit rechnen – er kann Differenzen bilden, inkrementieren etc. Daher eignen sich Zeiger gut für große und dynamische Speichermengen. Mit dem Adressoperator `&` ermitteln Sie die Adresse eines Objekts und erhalten einen Zeigertyp. Den erkennen Sie an einem dem Typ nachgestellten Stern `*`. Zum Beispiel können Sie nach `int x=12;` die Adresse mit `int* p = &x;` ermitteln. Dieser `int*` zeigt auf einen `int`-Wert, und ein `char*` zeigt auf einen `char`-Wert. Nur der Zeigertyp `void*` ist flexibel darin, worauf er zeigt – und deshalb unsicher und zu vermeiden. Zeigt ein Zeiger nirgendwo hin, dann hat er den speziellen Wert `nullptr`.

6.6.2 C-Arrays

C-Arrays werden mit eckigen Klammern `[]` notiert. Bei der Deklaration stehen diese hinter dem Variablennamen. Zum Beispiel speichert `int nums[10]` zehn `int`-Werte direkt nebeneinander. Der Typ von `nums` ist hier `int[10]` – solange `nums` nicht an eine Funktion übergeben wird. Leider kann der Compiler die Arraygröße nicht selbst mit übergeben, weswegen das C-Array dann zu einem Zeiger »verfällt«. So ist die größenlose Schreibweise `int[]` gleichbedeutend mit `int*`. Unter anderem wegen dieses großen Nachteils sollten Sie wenn möglich immer auf Alternativen ausweichen: `string`, `vector` oder Ähnliches für dynamische Datenmengen, `array`, `pair` oder `tuple` für fixe Mengen.

Kapitel 15
Vererbung

Kapiteltelegramm

- **»Hat-ein«-Beziehung**
  Entweder *Aggregation* oder *Komposition*; ein Objekt, das Teil eines anderen ist oder mit ihm in einer Beziehung steht; ein Auto *hat-ein* Lenkrad und *hat-eine* Garage.
- **»Ist-ein«-Beziehung**
  Vererbung; ein spezialisiertes Objekt ist auch ein allgemeineres Objekt; ein Bulli *ist-ein* Auto.
- **Komposition**
  Hat-ein-Beziehung, bei der das Objekt aus anderen Objekten besteht
- **Aggregation**
  Hat-ein-Beziehung, bei der das Objekt zu einem anderen Objekt in Beziehung steht
- **Überschreiben**
  Eine Methode einer Basisklasse in einer abgeleiteten Klasse mit gleicher Signatur neu definieren
- **Virtuelle Methode**
  Eine Methode, für die zur Laufzeit entschieden wird, welche überschriebene Variante aufgerufen wird
- **Slicing**
  Wenn Sie einen abgeleiteten Typ in einen Basistyp umwandeln und dabei Informationen verloren gehen; meist die Folge einer unbeabsichtigten Kopie des falschen Typs, zum Beispiel bei Call-by-Value
- **Laufzeitpolymorphie**
  Deklariert ist eine Basisklasse; zur Laufzeit wird eine abgeleitete Klasse verwendet, behält aber ihre abgeleiteten Eigenschaften.

Die Vererbung ist ein fundamentaler Bestandteil der *objektorientierten Programmierung* (OOP). Damit einher gehen bestimmte Techniken und ein Vokabular, beides stelle ich in diesem Kapitel vor.

Mit *Vererbung* können Sie zweierlei Dinge erreichen:

- Sie erhöhen die *Wiederverwendbarkeit* und reduzieren die *Codeduplikation* – das sind eher technische Aspekte.
- Sie implementieren ein *Design* und erzeugen so Klarheit in großen Projekten – dies ist ein konzeptioneller Aspekt.

Ich möchte Ihnen in diesem Buch vor allem die technischen Aspekte beibringen und werde nur kurz auf die konzeptionellen Aspekte eingehen. Sie sollten sie aber im Grundsatz kennen, damit Sie in Planungstreffen nicht aufgeschmissen sind.

15.1 Beziehungen

Beziehungen zwischen Objekten spielen eine zentrale Rolle in der objektorientierten Programmierung, da sie die Struktur und Funktionalität von Software maßgeblich beeinflussen und es ermöglichen, komplexe Systeme zu modellieren und zu organisieren. Dabei sind die Hat-ein- und Ist-Ein-Beziehungen von besonderer Bedeutung, da sie die Art und Weise definieren, wie Objekte miteinander verbunden sind und wie sie sich hierarchisch zueinander verhalten.

15.1.1 Hat-ein-Komposition

Als Sie in Kapitel 12, »Von der Struktur zur Klasse«, die Anwendung von `struct` und `class` kennengelernt haben, habe ich zunächst Datenfelder zu *Aggregaten* gebündelt.

```
class Auto {
    Lenkrad lenkrad_;
    std::vector<Rad> raeder_;
    // …
};
```

Dieses `Auto` *hat-ein* `lenkrad_` ebenso wie `raeder_` – jeweils vom entsprechenden Typ. In diesem Fall der Hat-ein-Beziehung ist es sogar aus diesen Objekten zusammengesetzt, bzw. besteht aus diesen Objekten. Das ist eine *Komposition*. In diesem Fall *besitzt* das Objekt meist seine Komponenten – was in C++ heißt, dass es für die Konstruktion und Destruktion verantwortlich ist. Die Lebenszeit der Komponenten ist durch die Lebenszeit des Objekts beschränkt.

Abbildung 15.1 Hat-ein-Beziehungen: die Komposition (links) und die Aggregation (rechts) in UML-Notation

15.1.2 Hat-ein-Aggregation

Anders ist die Situation, wenn Sie sagen, das Auto *hat-eine* Garage oder es *hat-einen* Eigentümer: Dies ist eine *Aggregation*.

```
class Auto {
    Garage& garage_;
    Person& eigentuemer_;
    // ...
};
```

Die enthaltenen Dinge stehen nur in einer *Beziehung* mit dem Objekt, sie *gehören* ihm aber nicht. Somit ist das Objekt im Regelfall auch nicht für deren Erzeugung oder Zerstörung verantwortlich. Die Lebenszeit der Elemente ist unabhängig von der Lebenszeit des Objekts.

Das anschauliche Beispiel von Auto, Lenkrad und Garage scheint offensichtlich jeweils Komposition und Aggregation zu beschreiben. Beim Design konkreter Software liegen die Dinge oft nicht ganz so einfach. Wenn Sie das Montageband eines Autoherstellers programmieren sollen, werden Sie in der Datenbank die Lenkrad-Instanzen anders in Beziehung zum Auto setzen, als wenn Sie ein Verzeichnis der Wagenflotte eines Autoverleihers anlegen sollen.

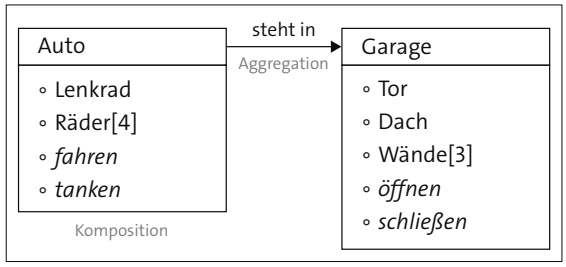

Abbildung 15.2 Eine alternative, mehr informelle Form, die beiden Hat-ein-Beziehungen darzustellen

In Abbildung 15.2 sehen Sie eine Darstellungsmöglichkeit der Beziehungen. Kompositionen sind gemeinsam in einem Kasten. Pfeile verbinden andere Zusammenhänge, Aggregationen werden mit durchgezogenen Linien und gefüllten Pfeilen dargestellt. Pfeile können zur Verdeutlichung noch die Art der Beziehung nennen. Innerhalb eines Kastens werden die Attribute (normale Schrift) und Verhalten/Methoden (*kursive Schrift*) aufgelistet.

Es gibt Dutzende verschieder Darstellungsarten für die Beziehungen zwischen Objekten. Ich habe hier nur einen ganz kurzen Einblick in zwei davon gegeben.

15.1.3 Ist-ein-Vererbung

Mit Vererbung bilden Sie nun eine *Ist-ein*-Beziehung ab. Ein passendes Beispiel wäre: Ein VW-Bulli *ist-ein* Auto. Das heißt:

- Ein VW-Bulli hat alle Eigenschaften, die auch ein Auto hat.
- Auf jedes Objekt, zu dem die Beschreibung (oder Spezifikation) eines VW-Bullis passt, passt auch die Beschreibung eines Autos.
- Die Beschreibung (oder Spezifikation) des VW-Bullis ist *genauer*, die des Autos hingegen *schwächer*.
- Wenn Sie ein Auto erwarten, dann ist es richtig, wenn Ihnen ein VW-Bulli geliefert wird.

Mit einem VW-Bulli und einem Auto ist es offensichtlich, dass Sie deren Beziehung durch Vererbung abbilden können. Doch manchmal sind die Dinge beim Zusammenstellen nicht ganz so eindeutig. Dann helfen diese Regeln vielleicht, zu überprüfen, ob eine *Ist-ein*-Beziehung vorliegt.

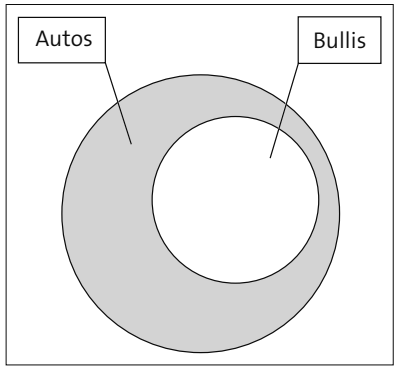

Abbildung 15.3 Dieses Venn-Diagramm zeigt, dass alle Bullis Autos sind, aber nicht alle Autos sind Bullis.

15.1.4 Ist-Instanz-von versus Ist-ein-Beziehung

Bitte beachten Sie, dass mit der »Ist-ein«-Beziehung für `Auto herbie;` nicht »`herbie ist-ein Auto`« gemeint ist. Dies nennt man »ist eine Instanz von« oder in C++ »`herbie istist vom Typ Auto`«. Es lässt sich nicht ganz vermeiden, dass die Begriffe nicht immer ganz korrekt verwendet werden. Zum Beispiel: »`7` ist eine Ganzzahl« oder »`7` ist ein `int`« ist in diesem Kontext nicht präzise. Aber immer »`7` ist Element der *Ganzen Zahlen*« oder »`7` ist vom Typ `int`« zu sagen, klingt seltsam.

Wenn Sie über Klassenhierarchien reden, sollten Sie aber auf diese Unterscheidung achten.

Nicht: ist-eine-Instanz-von

Merken Sie sich: Bei Vererbung ist mit »ist-ein« nicht die Instanz einer Klasse gemeint, sondern eine Unterklasse, die alle Eigenschaften der Oberklasse hat.

15.2 Vererbung in C++

In C++ bilden Sie Vererbung wie folgt ab:

```
class Auto {                        // Basis- oder Superklasse
public:
    Lenkrad lenkrad_;
    std::vector<Rad> raeder_;
    // …
};
class VwBulli : public Auto {       // VwBulli ist eine Unter- oder Subklasse
public:
    Dekoration bluemchen_;
    // …
};
```

Sie schreiben also den Namen der *Basisklasse* durch einen Doppelpunkt : und `public` getrennt hinter den Namen der aktuellen Klasse:

- `class` *Unterklasse* : `public` *Basisklasse* { …

Nun können Sie fröhlich neue `VwBulli`-Instanzen erzeugen. Und jede von ihnen hat automatisch auch ein `lenkrad_` und `raeder_`, obwohl Sie es in der Klasse `VwBulli` nicht mehr explizit erwähnt haben:

```
VwBulli vw{};
cout << vw.lenkrad_;
cout << vw.bluemchen_;
```

Wenn Sie ein pures `Auto auto;` erstellen, wird `auto.lenkrad_` ein korrekter Zugriff sein, aber ein `auto.bluemchen_` existiert nicht.

Wenn Sie also eine Instanz der Unterklasse haben, können Sie sowohl auf die eigenen Datenfelder und Methoden zugreifen als auch auf die von der Superklasse ererbten. Implementieren Sie zum Beispiel die Methode `campen` aus Abbildung 15.4 können Sie darin sowohl das `Lenkrad` als auch die `Blümchen` benutzen.

Andersherum geht das (logischerweise) nicht: Wenn Sie gerade `fahren` von `Auto` implementieren, dann können Sie zwar auf `Lenkrad` und die vier `Räder` zugreifen, die `Blümchen` existieren für Sie aber nicht.

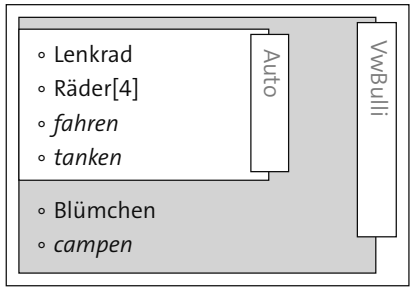

Abbildung 15.4 Alle Datenfelder und Methoden der Superklasse hat auch die Unterklasse.

15.3 Hat-ein versus ist-ein

Wie schon einleitend erwähnt, können Sie die Vererbungshierarchien logisch und konzeptionell designen. Wie, das hängt von der genauen Anwendung ab. Es kann mal sinnvoll sein, den VW-Bulli von Auto erben zu lassen, und in einer anderen Anwendung ist der VW-Bulli eine eigenständige Klasse, die ein Auto als Attribut hat.

Wenn Sie unsicher sind, ob Sie eine Vererbungshierarchie aufbauen sollten, ziehen Sie im Zweifel die Hat-ein-Beziehung der Ist-ein-Beziehung vor. In C++ ist Vererbung nur eine *Technik*, um das Design umzusetzen. Im Prinzip können Sie das Gleiche erreichen, indem Sie `Auto` zu einem Datenelement von `VwBulli` machen und die Methoden für `fahren` und `tanken` durchreichen.

Zugegeben, das ist an dieser Stelle starker Tobak. Sollten Sie irgendwann mal einige Klassen und Hierarchien erstellen, werden Sie sich vielleicht an diesen Tipp erinnern, dass Sie nicht alles in eine Vererbungshierarchie »quetschen« müssen. Manchmal tut es auch ein gut gekapseltes Datenfeld.[1]

15.4 Gemeinsamkeiten finden

Nachdem das gesagt ist, rudere ich auch gleich wieder vorwärts: An den Stellen, an denen Sie vielleicht auf eine logisch erscheinende Vererbungshierarchie verzichten, können Sie andererseits nur wenig verwandte Daten durch eine Vererbung zusammenführen und zum Beispiel Codeduplikation vermeiden. Das nenne ich, wie eingangs gesagt, den technischen Grund für eine Vererbungshierarchie.

Sehen Sie sich zum Beispiel Listing 12.24 und Listing 12.27 an. Wenn Sie die Datentypen `Month` und `Day` komplett wie `Year` ausschreiben, dann haben Sie eine Menge Code,

1 Sie verlieren dadurch möglicherweise die dynamische Typpolymorphie, doch ist statische Typpolymorphie mit Templates möglich.

der in drei Klassen identisch aussieht. Codeduplikation ist schlecht, und daher können Sie eine kleine Vererbungshierarchie aufbauen, um diese zu vermeiden.

Zunächst definieren Sie in Listing 15.1 den gemeinsamen Vorfahren `Value` der drei Hilfsklassen:

```
// https://godbolt.org/z/hhGo46z5a
#include <iostream>  // ostream
#include <format>    // format, vformat, make_format_args
using std::ostream;
class Value {
protected: // nicht öffentlich, nur für den eigenen und abgeleiteten Gebrauch
    int value_;
    const std::string fmt_;  // z. B. "{:02}" oder "{:04}"
    Value(int v, unsigned w) // Konstruktor mit zwei Argumenten
      : value_{v}, fmt_{std::format("{{:0{}}}", w)} {}
public:
    ostream& print(ostream& os) const;
};
ostream& operator<<(ostream& os, const Value& rechts) {
    return rechts.print(os);
}
ostream& Value::print(ostream& os) const {
    return os << std::vformat(fmt_, std::make_format_args(value_));
}
```

Listing 15.1 Der gemeinsame Vorfahre unserer Hilfsklassen »Year«, »Month« und »Day«

Hiermit haben Sie dann alle wichtigen Funktionen der drei Wertklassen implementiert. Ich habe das Datenfeld `fmt_` hinzugefügt, weil `Year` mit einer Breite von vier ausgegeben werden wird, `Month` und `Day` dagegen mit einer Breite von zwei. Die variable Wunschbreite der Ausgabe haben Sie als Parameter dem Konstruktor hinzugefügt. Das Format `fmt_` für den `vformat`-Aufruf in `print` wird im Konstruktor vorbereitet.

Alternativ hätten Sie auch einen `formatter` für `Value` oder `Date` schreiben können, das wäre aber mehr Aufwand gewesen.

Die Deklaration der drei Hilfsklassen ist nun sehr kurz, wie Sie in Listing 15.2 sehen können:

```
// https://godbolt.org/z/eYY1js4Te
class Year : public Value {                  // von Klasse Value ableiten
public:
    explicit Year(int v) : Value{v, 4} {} // Basisklasse initialisieren
};
```

```
class Month : public Value {
public:
    explicit Month(int v) : Value{v, 2} {}
};
struct Day : public Value {                 // class-public entspricht struct
    explicit Day(int v) : Value{v, 2} {}
};
```

Listing 15.2 Der doppelte Code der Hilfsklassen ist nun verschwunden.

Es bleiben nur die Konstruktoren des jeweiligen Datentyps übrig, alles andere wird von der ererbten Klasse `Value` erledigt.

Beachten Sie insbesondere die folgenden Punkte:

- Bei `class Year : public Value` sage ich mit `: public Value`, dass ich diese neue Klasse von der Klasse `Value` ableite.
- Ich rufe jeweils in der Initialisierungsliste der Konstruktoren den Konstruktor von `Value` mit zwei Argumenten auf, zum Beispiel bei `Year(int v) : Value{v, 4}`… Die Basisklasse muss ja auch initialisiert und einer ihrer Konstruktoren *muss* aufgerufen werden. Welcher das ist, legen Sie hinter dem Doppelpunkt `:` der Unterklasse fest. Lassen Sie diesen expliziten Aufruf weg, versucht es der Compiler mit dem Konstruktor der Basisklasse ohne Argumente. Das wäre hier `Value{}` gewesen, den es aber nicht gibt.
- `class Value` beginnt mit einem `protected`-Bereich. Das heißt, nur die Klasse selbst und abgeleitete Klassen dürfen auf den Inhalt zugreifen, aber nicht von außen (`public`). Der Konstruktor ist in diesem Bereich. Dadurch können die Unterklassen den Konstruktor in ihren Initialisierungen als Teil des Konstruktors aufrufen, aber ich kann zum Beispiel nicht in `main` einen `Value val{10,3};` definieren – das wäre ein Zugriff auf den Konstruktor »von außen«.
- Der Basiskonstruktor `Value(int v, unsigned w)` muss nicht `explicit` sein, denn mit zwei Argumenten ist er kein Kandidat mehr, um zur automatischen Typumwandlung herangezogen zu werden. Ein Fehler wäre es nicht gewesen, aber überflüssig.
- Zur Erinnerung noch einmal: Einen neuen Typ mit `class` zu deklarieren und sofort einen `public:`-Bereich anzufangen, ist so als hätten Sie ihn sofort mit `struct` definiert. Ich habe das exemplarisch mit `struct Day` einmal gemacht. Was Sie wählen, ist Geschmackssache, eine Richtlinie zu haben, sinnvoll.

Der Rest des Listings bleibt größtenteils erhalten. `Date` verwendet `Year`, `Month` und `Day` wie gehabt. Für `Date` ist die Veränderung unsichtbar geblieben – ein Vorteil der Kapselung.

```
// https://godbolt.org/z/9rY7qhK89
class Date {
    Year year_;
    Month month_ {1};
    Day day_ {1};
public:
    explicit Date(int y) : year_{y} {} // year-01-01
    Date(Year y, Month m, Day d) : year_{y}, month_{m}, day_{d} {}
    ostream& print(ostream& os) const;
};
ostream& Date::print(ostream& os) const {
    return os << year_ << "-" << month_ << "-"  << day_;
}
ostream& operator<<(ostream& os, const Date& rechts) {
    return rechts.print(os);
}
int main() {
    Date d1 { Year{2013}, Month{15}, Day{19} };
    std::cout << d1 << "\n"; // Ausgabe: 2013-15-19
}
```

Listing 15.3 So verwendet »Date« die neuen Klassen.

15.5 Abgeleitete Typen erweitern

In Listing 12.24 war `ostern` eine freie Funktion. Würde es nicht Sinn ergeben, wenn `ostern` eine Methode von `Year` wäre? Dann könnten wir eine Instanz `Year year{2018}` direkt mit `year.ostern()` fragen, auf welchen Tag Ostern in dem Jahr fällt.

```
// https://godbolt.org/z/Wjsvv6Gdv
class Date;  // Vorwärtsdeklaration
class Year : public Value {
public:
    explicit Year(int v) : Value{v, 4} {}
    Date ostern() const;            // neue Methode deklarieren
};
// Hier Month, Day und Date deklarieren. Dann:
Date Year::ostern() const {         // neue Methode definieren
    const int y = value_;
    int a = value_/100*1483 - value_/400*2225 + 2613;
    int b = (value_%19*3510 + a/25*319)/330%29;
    b = 148 - b - (value_*5/4 + a - b)%7;
    return Date{Year{value_}, Month{b/31}, Day{b%31 + 1}};
}
```

```
int main() {
    using std::cout;
    Year year{2014};
    cout << year.ostern() << "\n";  // Ausgabe: 2014-04-20
}
```

Listing 15.4 Nun ist »ostern« eine Methode von »Year«.

Weil ich `Date` in der Deklaration von `Year::ostern()` in `Date ostern() const;` schon verwende, bevor es definiert wurde, muss ich ganz zu Beginn mit `class Date` bekannt machen, dass es einen solchen Typ geben wird. Vor der wirklichen Verwendung bei der Definition von `Year::ostern()` ab `Date Year::ostern() const {…` muss der Typ dann aber definiert worden sein – `class Date { … };` muss davor stehen.

Bisher hatten unsere drei Hilfsklassen nicht mehr Funktionalität als `Value`. Nun habe ich `Year` um eine Methode erweitert. Auch haben `Day` und `Month` diese Methode nicht – `Year` ist nun, was die Implementierung angeht, wirklich unterschiedlich.

Wiederverwendung und Erweiterung sind beides elementare Konzepte der objektorientierten Programmierung.

15.6 Methoden überschreiben

In der Klasse `Value` gab es in keiner der abgeleiteten Klassen Methoden, die genau gleich hießen bzw. die gleiche *Signatur* hatten. Die Signatur einer Funktion (oder Methode) ist durch den Namen und die Parametertypen (inklusive eines eventuellen `const` für `this`) festgelegt.

Sie können in einer abgeleiteten Klasse durchaus eine Methode ebenso nennen wie in der Basisklasse. Dies nennt man dann *Überschreiben* der Methode. So können Sie in größeren Hierarchien Standardverhalten in der Basisklasse vordefinieren und in den Klassen neu definieren, die »etwas anders« sind:

```
struct Komponente {
    Color getColor() const { return weiss; }
};
struct Fenster : public Komponente { };
struct Hauptfenster : public Fenster { };
struct Dialog : public Fenster { };
struct Texteingabe : public Komponente { };
struct Druckknopf : public Komponente {
    Color getColor() const  { return grau; }
};
```

Listing 15.5 Alle Komponenten haben eine weiße Farbe, nur der Druckknopf wird grau.

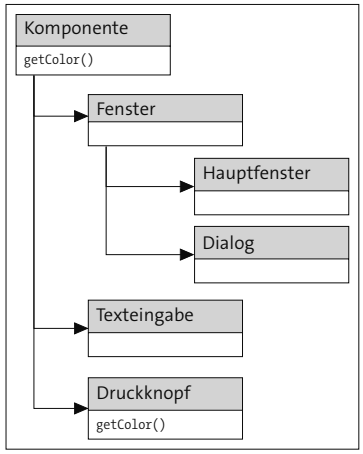

Abbildung 15.5 Eine Klassenhierarchie mit einer überschriebenen Methode

Wenn Sie nun `Komponente k{};` `k.getColor();` schreiben, werden Sie ebenso `weiss` bekommen wie für `Dialog d{};` `d.getColor();` etc. Nur `Druckknopf kn{};` `kn.getColor();` liefert `grau` zurück.[2] Falls Sie sich die Klassenhierarchie aus Listing 15.5 nicht erschließen können, finden Sie in Abbildung 15.5 eine alternative Darstellung.

15.7 Wie Methoden funktionieren

Wenn Sie in Klassenhierarchien Methoden verwenden, müssen Sie sich einiger Dinge bewusst sein. Schauen Sie sich einmal Listing 15.6 an: Achten Sie darauf, welche anderen Methoden `print` aufruft, und überlegen Sie, was Sie erwarten würden. Ich habe das Programm auf das Wesentliche reduziert, weswegen es etwas theoretisch aussieht.

```
// https://godbolt.org/z/476G1xzKr
#include <iostream>
struct Basis {
    int acht_ = 8;
    int wert() const { return acht_; }
    void print(std::ostream& os) const { os << wert() << "\n"; }
};
struct Print : public Basis {
    int neun_ = 9;
    void print(std::ostream& os) const { os << wert() << "\n"; }
};
```

2 In der Praxis wird eine solche Komponentenhierarchie so nicht vorkommen, es fehlt noch das Schlüsselwort `virtual`, das ich etwas später beschreibe.

```
struct Wert : public Basis {
    int zehn_ = 10;
    int wert() const { return zehn_; }
};
struct Beides : public Basis {
    int elf_ = 11;
    int wert() const { return elf_; }
    void print(std::ostream& os) const { os << wert() << "\n"; }
};
int main() {
    Basis ba{}; ba.print(std::cout);   // Basisaufruf
    Print pr{}; pr.print(std::cout);   // print überschrieben
    Wert we{}; we.print(std::cout);    // print aus Basis
    Beides be{}; be.print(std::cout);  // alles überschrieben
}
```

Listing 15.6 Was gibt »print« aus? Die Methode »wert« kommt öfter vor.

Was geben die unterschiedlichen `print`-Zeilen in `main` Ihrer Meinung nach aus?

- **`Basis ba{}; ba.print(cout);` – Basisaufruf**

  In `Basis` sind die beiden Methoden `print` und `wert` direkt definiert. Hier ist noch keine andere Klasse beteiligt. `wert` liefert also `acht_` zurück, und `8` wird ausgegeben.

- **`Print pr{}; pr.print(cout);` – `print` überschrieben**

  In `Print` ist die Methode `print` *überschrieben*. Für `pr` wird `Print::print` aufgerufen. Dort wird `wert` benötigt. Diese Methode wurde von `Basis` geerbt. Und `Basis::wert` liefert `acht_` zurück – also wird `8` ausgegeben.

- **`Wert we{}; we.print(cout)` – `print` aus `Basis`**

  `we.print` muss auf die ererbte Methode `Basis::print` zurückgreifen. Dort wird nun die Methode `wert()` benötigt. Obwohl es die Methode `Wert::wert` gibt, ist sie in `Basis::print` aber *noch nicht bekannt*! In einer Methode der Basisklasse werden auch nur Methoden der Basisklasse verwendet, hier also `Basis::wert`, was `acht_` zurückgibt und zur Ausgabe von `8` führt.

- **`Beides be{}; be.print(cout);` – alles überschrieben**

  Hier ist es wieder einfach: Es gibt ein `Beides::print`, das aufgerufen wird. Der Aufruf von `wert` greift in der eigenen Methode `Beides::wert` auf `elf_` zurück. Es wird `11` ausgegeben.

Haben Sie den dritten Fall richtig »erraten«? Wenn nicht, ärgern Sie sich nicht. Welche `wert()`-Methode von `Basis::print` genommen wird, ist eine Frage der Definition

– beides ist möglich. In C++ ist es so definiert, dass eine Methode nur die Methoden sieht, die zum Zeitpunkt der Übersetzung der Klasse per Vererbung, neuer Definition oder Überschreiben zur Verfügung stehen. `Basis::print` weiß von eventuell abgeleiteten Klassen noch nichts.

15.8 Virtuelle Methoden

In anderen Sprachen ist das teilweise anders: Hätten Sie obiges Listing mit offensichtlichen Veränderungen in Java geschrieben, dann hätten Sie bei `Wert we{}; we.print(cout)` eine `10` gesehen – Java schaut zur *Laufzeit* nach, welche Methoden einer Instanz zur Verfügung stehen.

Und weil das ebenfalls ein sinnvolles Verhalten ist (sonst hätte man dies als Standardverhalten in Java wohl kaum gewählt), ist es auch in C++ möglich. Der Schlüssel dazu sind *virtuelle Methoden*.

Wenn Sie eine Methode mit dem Schlüsselwort `virtual` markieren, entscheidet der Compiler zur *Laufzeit*, welche Version dieser Methode gültig ist. Sie erhalten also das Java-Verhalten.

```
// https://godbolt.org/z/aj8cnox4f
#include <iostream>

using std::ostream; using std::cout;
struct Basis2 {
    int acht_ = 8;
    virtual int wert() const          // virtuelle Methode
        { return acht_; }
    void print(ostream& os) const
        { os << wert() << "\n"; }
};
struct Wert2 : public Basis2 {
    int zehn_ = 10;
    virtual int wert() const override // überschreiben
        { return zehn_; }
};
int main() {
    Wert2 we2{}; we2.print(cout);     // verwenden
}
```

Listing 15.7 Mit »virtual« markierte Methoden werden zur Laufzeit aufgelöst.

`Wert2 we2{}; we2.print(cout);` entspricht dem Aufruf `Wert we{}; we.print(cout);` aus dem vorigen Listing. Nun werden Sie eine `10` zu Gesicht bekommen. `we2.print()` muss zwar auf die Definition `Basis2::print` zurückgreifen, denn `Wert2` hat diese Methode nicht selbst definiert, aber der Aufruf von `wert()` in `print` ist nun ein *virtueller Methodenaufruf* – er wird zur Laufzeit entschieden. Und da `we2` vom Typ `Wert2` ist, wird `Wert2::wert` verwendet, `zehn_` zurückgegeben und somit `10` ausgegeben.

Virtuell und nicht virtuell

Virtuelle Methodenaufrufe werden zur Laufzeit entschieden, normale Methodenaufrufe zur Übersetzungszeit.

In Listing 15.7 sehen Sie bei der Definition der Methode `wert` zusätzlich zu dem `virtual` an der Methode auch noch ein `override`. Durch eine solche zusätzliche Auszeichnung verlangt der Compiler, dass diese Methode auch wirklich eine andere Methode überschreibt. So können Sie zum Beispiel vermeiden, durch einen Tippfehler schwer zu findende Fehler zu produzieren. Stellen Sie sich zum Beispiel vor, Sie hätten versehentlich in `Wert2` die Methode `virtual int wetr() const` genannt. Das Programm wird kompilieren und laufen, aber nicht die Werte zurückliefern, die Sie gerne hätten.

Oder schauen Sie sich dieses andere, durchaus praxisnahe Beispiel an. In einem Header `"mydefs.hpp"` sind einige Typen definiert, zum Beispiel `using value_t = int;`. Und nun haben Sie eine kleine Hierarchie:

```
struct Number {
    virtual void add(value_t value);
};

struct SafeNumber : public Number {
    virtual void add(int value);      // override nicht angegeben, int statt value_t
};
```

Dass in `Number::add` der Parameter vom Typ `value_t` ist, aber in `SafeNumber::add` als Typ `int` angegeben wurde, ist für den Compiler das Gleiche. Bei `using` (und das entsprechende `typedef`) handelt es sich nur um einen Typalias, aber nicht um einen komplett neuen Typ. `SafeNumber::add` überschreibt daher wie gewünscht den Vorgänger.

Wenn Sie in `"mydefs.hpp"` nun die Definition auf `using value_t = long;` ändern, dann ändert sich die Signatur von `Number::add`. Die Signatur von `SafeNumber::add` ändert sich aber nicht, weil Sie nicht den Typalias verwendet haben. Sie überschreiben die Ursprungsmethode nicht mehr und bekommen sehr wahrscheinlich ein unerwünschtes Verhalten Ihres Programms – einen schwer zu findenden Fehler. Wenn Sie

`SafeNumber::add` mit `override` versehen hätten, dann hätte der Compiler Sie auf den Fehler hingewiesen.

Verwenden Sie »override«

Wenn Sie eine virtuelle Methode überschreiben, geben Sie zusätzlich auch `override` mit an. Sie vermeiden auf diese Weise schwer aufzufindende Fehler.

15.9 Konstruktoren in Klassenhierarchien

Konstruktoren sind besondere Klassenelemente: Sie sind *keine* Methoden, und daher werden sie nicht wie Methoden an abgeleitete Klassen weitervererbt.

```
// https://godbolt.org/z/57vexTPnn
class Base {
public:
    Base() {}                 // null-Argument-Konstruktor
    explicit Base(int i) {}   // ein Argument
    Base(int i, int j) {}     // zwei Argumente
    void func() {};           // Methode
};

class Derived : public Base { // kein eigener Konstruktor
};

int main() {
    Base b0{};                // okay, null-Argument-Konstruktor
    Base b1{12};              // okay, ein Argument
    Base b2{6,18};            // okay, zwei Argumente
    Derived d0{};             // okay, Compiler generiert Defaultkonstruktor
    d0.func();                // okay, Methode wird geerbt
    Derived d1{7};            // ↯ Fehler: kein Konstruktor für ein Argument
    Derived d2{3,13};         // ↯ Fehler: kein Konstruktor für zwei Argumente
}
```

Listing 15.8 Die abgeleitete Klasse erbt Methoden, aber nicht Konstruktoren der Elternklasse.

Der Versuch, mit `Derived d1{7}` ein neues Objekt anzulegen, schlägt also fehl, weil `Derived` keinen eigenen Konstruktor für einen `int` definiert. Der `Base(int)`-Konstruktor wird nicht weitervererbt, wie dies bei normalen Methoden der Fall ist – beispielsweise bei `func()`, die Sie auch für `Derived`-Instanzen aufrufen können.

Der Grund dafür ist, dass Konstruktoren wirklich etwas anderes sind als Methoden. Ein Konstruktor muss eine Klasseninstanz initialisieren. Das ist eine Aufgabe, die eng an die innere Struktur der Klasse gekoppelt ist. Einen Konstruktor der Elternklasse statt eines eigenen Konstruktors aufzurufen, kann sehr wahrscheinlich gar keine korrekte Initialisierung der abgeleiteten Klasse vollbringen. Zu einer Instanz kann es Verwaltungsinformationen geben, die mit der Klasse zu tun haben und die im jeweiligen Konstruktor initialisiert werden müssen. Welche Verwaltungsinformationen das sind, ist nicht bis ins letzte Detail vom Standard festgelegt, es wird der Implementierung überlassen. Um diese Freiheit zu erlauben, haben Konstruktoren diese besondere Rolle.

Wenn Sie dennoch die Konstruktoren der Elternklasse haben wollen, müssen Sie dies in der abgeleiteten Klasse explizit sagen. Dafür fügen Sie in die Klassendefinition `using Base::Base;` ein, erwähnen also den genauen Namen der Konstruktoren. Ja, *der Konstruktoren*, denn mit diesem Mechanismus erheben Sie alle ererbten Konstruktoren auf einmal, nicht einen einzelnen. Diesen Mechanismus können Sie nur nach dem Motto »alles oder nichts« verwenden.

```
// https://godbolt.org/z/8b3Yvnjh4
class Base {
public:
    Base() {}
    explicit Base(int i) {}
    Base(int i, int j) {}
    void func() {};           // Methode
};

class Derived : public Base {
public:
    using Base::Base;         // Importieren aller Konstruktoren der Elternklasse
};

int main() {
    Derived d0{};             // okay, importiert, nicht mehr generiert
    Derived d1{7};            // okay, wurde importiert
    Derived d2{3,13};         // okay, wurde importiert
}
```

Listing 15.9 Mit »using« importieren Sie alle Konstruktoren der Elternklasse.

Auf diese Weise erheben Sie alle Konstruktoren der Elternklasse in die eigene Klasse, und die nötigen zusätzlichen Verwaltungsaufgaben übernimmt der Compiler.

15.10 Typumwandlung in Klassenhierarchien

Bleiben wir bei unserem abstrakten Beispiel, aber lassen Sie es mich ein wenig um gefährlichen Code erweitern.

15.10.1 Die Vererbungshierarchie aufwärts umwandeln

```
// https://godbolt.org/z/jfocr7qf1
//... Basis2 und Wert2 wie gehabt ...
void ausgabe(Basis2 x) {          // Übergabe als Wert
    x.print(cout);
}
int main() {
    Basis2 ba2{}; ausgabe(ba2); // gibt 8 aus
    Wert2 we2{}; ausgabe(we2);  // gibt auch 8 aus
}
```

Listing 15.10 Die Übergabe als Wert kopiert nur den gemeinsamen Teil des Typs.

Durch die Übergabe als Wert wird `we2` in den Parameter `x` *kopiert*. Weil der Parameter vom Typ `Basis2` ist, wird auch nur jener Teil von `we2` kopiert, der zu `Basis2` gehört. Und weil `x` nun von diesem Typ ist, kann `x.print` nur das tun, was jede andere Variable von diesem Typ in `Basis2::print` tun würde – `8` ausgeben.

15.10.2 Die Vererbungshierarchie abwärts umwandeln

Hätten Sie für den Parameter nicht den Basistyp gewählt, sondern den abgeleiteten – also `ausgabe(Wert2 x)` –, dann hätte `ausgabe(we2)` die `10` ausgegeben. Jedoch passt `ba2` nicht auf den Parametertyp `Wert2`: Die Umwandlung in diese Richtung der Hierarchie quittiert der Compiler mit einem Fehler. Instanzen vom Basistyp haben nicht alle Eigenschaften, die nötig sind, um in den abgeleiteten Typ umgewandelt zu werden – der Compiler kann sich ja keine Eigenschaften zum Auffüllen ausdenken.

```
// https://godbolt.org/z/j9Er3s1xo
//... Basis2 und Wert2 wie gehabt ...
void ausgabe(Wert2 x) {           // abgeleitete Klasse als Wert
    x.print(cout);
}
int main() {
    Basis2 ba2{}; ausgabe(ba2); // ↯ ba2 kann nicht in Wert2 umgewandelt werden
    Wert2 we2{}; ausgabe(we2);  // gibt 10 aus
}
```

Listing 15.11 Die abgeleitete Klasse als Argumenttyp zu einer Funktion erlaubt nicht den Aufruf mit einer Variable der Basisklasse als Wertparameter.

15.10.3 Referenzen behalten auch die Typinformation

Wenn Sie eine Instanz als Referenz übergeben, muss sie nicht kopiert werden. In diesem Fall bleibt das Objekt als solches erhalten – und mit ihm zusammen dessen eigentlicher Typ.

```
// https://godbolt.org/z/3nvrvrrYW
//... Basis2 und Wert2 wie gehabt ...
void ausgabe(Basis2& x) {          // Übergabe als Referenz
    x.print(cout);
}

int main() {
    Basis2 ba2{}; ausgabe(ba2); // gibt 8 aus
    Wert2 we2{}; ausgabe(we2);  // gibt 10 aus, denn das Objekt wird nicht kopiert
}
```

Listing 15.12 Die Übergabe als Referenz verändert die Instanz nicht.

Wenn `we2` zu `x` wird, dann wird es nur »umbenannt«. `x` bleibt im Kern ein `Wert2` – somit kann `print` auf die virtuelle Methode `Wert2::wert` zugreifen und `10` ausgeben.

Man nennt es *Laufzeitpolymorphie*, wenn man eine allgemeinere Klasse in einer Deklaration verwendet, zur Laufzeit aber eine andere konkretere Klasse für die deklarierte Variable eingesetzt wird, ohne dass diese ihre Eigenschaften verliert. Im Beispiel: Obwohl der Parameter als `Basis2` deklariert ist, kann `ausgabe` zur Laufzeit mit einer Instanz `we2` der Klasse `Wert2` aufgerufen werden, *und* `we2` behält seine Eigenschaft bei, `10` auszugeben – was es nicht täte, wäre es komplett in den Typ `Basis2` umgewandelt worden. Diese Form der Polymorphie erhalten Sie in C++ nur, wenn Sie mit Referenzen (oder Zeigern) arbeiten.

Falls die Frage auftaucht: Wäre `Basis2::wert` nicht virtuell (also wie `Basis::wert`), wären Sie wieder beim Verhalten aus Listing 15.6 und erhielten eine `8`.

15.11 Wann virtuell?

Ob Sie eine Methode mit `virtual` versehen oder nicht, hängt davon ab, für welches *Design* Sie sich entscheiden. Es ist in C++ nicht üblich, pauschal alle Methoden aller Klassen virtuell zu machen. Für jede einzelne Methode kann es Gründe dafür und dagegen geben, für jede einzelne Klasse ebenfalls. Das ist hier ein wenig anders als bei den Gründen für die Wahl zwischen Methoden und freien Funktionen: Die Kapselung mit Methoden ist besser, und so entscheidet man sich im Normalfall für diese. Bei der Frage, ob Sie virtuelle Methoden einsetzen sollten oder nicht, ist die Sache

nicht ganz so klar. Nehmen Sie die folgenden Hinweise als Entscheidungshilfen für Ihr Design:

- Manche Klassen dienen hauptsächlich dem Zweck, Daten zusammenzuhalten. Sie sind nicht Teil einer Hierarchie. Ohne Vererbung gibt es keinen Grund für virtuelle Methoden.
- Die Existenz einer Klassenhierarchie alleine rechtfertigt noch keine virtuellen Methoden. Vielleicht deduplizieren Sie nur sehr geschickt.
- Innerhalb einer Klassenhierarchie eine Methode zu überschreiben, ist ein guter Kandidat für eine virtuelle Methode, jedoch nicht zwangsläufig.
- Wenn Ihr Design auf überschriebene Methoden baut, also auf sich änderndes Verhalten von Klasse zu Klasse, ist `virtual` angesagt.
- Konstruktoren sind niemals virtuell. Innerhalb von Konstruktoren dürfen Sie keine virtuellen Methoden aufrufen.
- Ein Destruktor (siehe nächstes Kapitel) muss virtuell sein, wenn es mindestens eine virtuelle Methode in der Klasse gibt.
- Eine virtuelle Methode einer Basisklasse, die Sie überschreiben, ist automatisch ebenfalls virtuell, auch wenn Sie es nicht explizit sagen – »einmal `virtual`, immer `virtual`«.

Klassen entweder ganz ohne oder ganz mit »virtual«

Für den Anfang empfehle ich Ihnen, diesen zwei Regeln zu folgen:

- Ohne Hierarchie machen Sie nichts `virtual`.
- Mit Hierarchie machen Sie *alle* Methoden `virtual`, wenn Sie mindestens eine Methode überschreiben.

Wenn Sie das im Hinterkopf haben, werden Sie vielleicht schon ganz von selbst die Klassen(-Hierarchien) so entwerfen, dass die Entscheidung, ob eine Methode virtuell sein soll oder nicht, ganz von selbst fällt.

Doch warum überhaupt die Frage? Was sind die Vor- und was die Nachteile von virtuellen Methoden?

Der große Vorteil ist, dass Sie ein flexibleres Design haben. Sie können in abgeleiteten Klassen Verhalten verändern, das in einer Basisklasse eigentlich schon festgelegt war.

Die Nachteile betreffen Geschwindigkeit und Speicher:

- **Die Methode muss zur Laufzeit erst in einer Tabelle nachgeschlagen werden.**

  Wenn Sie eine virtuelle Methode aufrufen, dann benötigt dieser Aufruf eine Indirektion (und möglicherweise eine Addition) mehr: Moderne Prozessoren erledi-

gen dies jedoch aus dem Effeff, und Sie werden keine Geschwindigkeitunterschiede zu einem normalen Methodenaufruf bemerken.

- **Virtuelle Methoden können selten zu Inline-Funktionen optimiert werden.**

  Dies könnten Sie eher bei der Geschwindigkeit des Programms bemerken, aber nur, wenn Sie eine virtuelle Funktion in einer engen Schleife aufrufen. Inlining ist eine wichtige Optimierung des Compilers, die desto wichtiger ist, je moderner und komplexer der Prozessor ist, der das Programm abarbeitet.

- **Pro Instanz wird ein verstecktes Datenfeld benötigt.**

  Jede Instanz einer Klasse mit mindestens einer virtuellen Methode hat eine zusätzliche »versteckte« Variable. So ist das zumindest in den meisten C++-Compilern implementiert: Es handelt sich um einen Zeiger (`vptr`) in eine statische Tabelle (`vtable`). Wenn Sie kleine Instanzen haben, die Sie in großen Mengen zum Beispiel in einen Container packen, dann wirkt sich das beim Speicher aus.

Sie könnten auch argumentieren, dass Sie sich beim Design an anderen aktuellen Programmiersprachen orientieren möchten. Java zum Beispiel besitzt gar keine nicht virtuellen Methoden. Ich plädiere mindestens dafür, *datenhaltende Klassen* (*Data Transfer Objects*, DTOs) von *verhaltensorientierten Klassen* (mit nicht trivialen Methoden) im Design zu trennen: DTOs benötigen keine Virtualität, echt rechnende Klassen je nach Zweck vielleicht. Jemand, der sich »zur Sicherheit« dafür entscheidet, *alle* Methoden virtuell zu machen, den würde ich nur fragen, ob das für seine Anwendung angemessen ist, ansonsten würde ich diese Designentscheidung verstehen.

15.12 Andere Designs zur Erweiterbarkeit

Wenn Sie sich dafür entscheiden, keine virtuellen Methoden einzusetzen, gibt es in C++ andere Wege, um Datentypen dennoch erweiterbar zu machen. Sie können durch die Überladung freier Funktionen Funktionalität hinzufügen. Ein fortgeschrittenes Thema ist die Verknüpfung von Typen anhand von *Type-Traits* – dazu müssen Sie noch etwas über Templateprogrammierung erfahren. Sie finden einen kurzen Einstieg in Kapitel 23, »Templates«.

Verwenden Sie virtuelle Methoden ruhig und gerne auch großzügig. Sie bieten einen guten Kompromiss zwischen Verständlichkeit und Performance. Wenn Sie eines Tages mit Spezialanwendungen zu tun haben, können Sie sich immer noch andere Möglichkeiten aneignen.

Etwas Generelles noch zur Vererbung: Üben Sie ruhig mit der Technik des Vererbens, nutzen Sie es zur Reduktion von Codeduplikation. Strapazieren Sie das Konzept aber nicht über. Sehr viel häufiger als die *Ist-ein*-Beziehung – Vererbung – eignet sich die *Hat-ein*-Beziehung – Komposition oder Aggregation –, um Klassen und Objekte sinn-

voll zu verbinden. Zwängen Sie nicht »auf Teufel komm raus« eine Menge von Klassen in das starre Korsett einer Vererbungshierarchie. In der Praxis stellen sich Designs als flexibler heraus, wenn diese mehr über Datenfelder erweitert und modifiziert werden können als über eine Klassenhierarchie.

Während Sie die Komposition in C++ mit der Überladung freier Funktionen oder Type-Traits abbilden können, so sind dafür aus Sprachen wie Java die *Interfaces* bekannt. Die Entsprechung in C++ sind abstrakte Klassen nur mit pur virtuellen Methoden, siehe Kapitel 16, »Der Lebenszyklus von Klassen«. Diese Technik ist in C++ zwar auch üblich, wird aber bei Weitem nicht so breit eingesetzt wie in Java.

Kapitel 17
Guter Code, 4. Dan: Sicherheit, Qualität und Nachhaltigkeit

In diesem Kapitel will ich Ihnen einige Dinge zusammengefasst auflisten, die Ihnen besonders bei der Entwicklung *guter* Software helfen können. Im Buch verteilt finden Sie noch viele weitere Tipps, und insbesondere in Kapitel 29, »Threads – Programmieren mit Mehrläufigkeit«, eine lange Sammlung, jedoch jeweils nur kurz angerissen.

Das Besondere an den Tipps dieses Kapitels ist, dass sie am besten *von Anfang an* angewendet werden. Wenn Sie die Idee *hinter* den Regeln in Ihren täglichen Programmieralltag aufnehmen, werden sich Ihre Sicht auf C++ und Ihr Denken mit C++ so verändern, dass automatisch besserer Code entsteht.

17.1 Die Nullerregel

Eine Klasse kann mit vielen Funktionen für besondere Einsätze ausgerüstet werden. Sie können eine automatische Konvertierung veranlassen, Sie können sagen, wie eine Klasse kopiert werden soll etc.

Lesen Sie die folgenden Absätze entspannt, denn am Ende werde ich Ihnen sagen, dass Sie keine dieser Funktionen implementieren sollen.

17.1.1 Die großen Fünf

Besonders auf die folgenden Funktionen müssen Sie achten:

- **Destruktor `~Typ()`**

  Der Computer generiert Ihnen einen Destruktor, der Membervariablen wegräumt, aber nicht für rohe Zeiger `delete` oder C-Arrays `delete[]` aufruft (siehe Kapitel 20, »Zeiger«). Hat Ihre Klasse solche Felder, müssen Sie einen Destruktor schreiben.

- **Kopierkonstruktor `Typ(const Typ&)`**

  Wenn Sie nicht selbst einen Kopierkonstruktor schreiben, dann erzeugt der Compiler einen. Dieser kopiert aber nur alle Felder. Das ist zum Beispiel bei rohen Zeigern schlecht, weil dann zwei Objekte auf das gleiche Speicherobjekt zeigen – es sieht so aus, als würden beide das Objekt »besitzen«. Und wenn Sie (richtigerweise)

einen eigenen Destruktor geschrieben haben, rufen beide `delete` auf demselben Speicherobjekt auf. Sie müssen selbst einen Kopierkonstruktor schreiben, der den *Inhalt* von Zeigern kopiert.

- **Zuweisungsoperator `Typ& operator=(const Typ&)`**

  Wenn Sie den Zuweisungsoperator nicht selbst schreiben, aber rohe Zeiger zur Klasse gehören, dann generiert der Compiler eine Zuweisung, die aktuelle Zeiger einfach überschreibt, ohne sie vorher zu löschen. Zusätzlich zeigen nach der Zuweisung beide Objekte auf denselben Speicherbereich: doppelt böse. Daher müssten Sie mit rohen Zeigern hier selbst eingreifen und wieder den Inhalt kopieren.

- **Verschiebekonstruktor `Typ(Typ&&)`**

  Mit rohen Zeigern generiert der Compiler (wenn überhaupt) das Gleiche wie beim Kopierkonstruktor. Wollen Sie verschieben, müssen Sie dies implementieren.

- **Verschiebeoperator `Typ& operator=(Typ&&)`**

  Ebenso wie beim Verschiebekonstruktor behandelt ein eventuell vom Compiler erzeugter Verschiebeoperator rohe Zeiger nicht zufriedenstellend. Sie müssen selbst Hand anlegen.

Man sagt, wenn Sie auch nur eine einzige dieser fünf Funktionen implementieren müssen, dann müssen Sie auch die anderen implementieren. Dies ist die »Rule of Five« (Fünferregel). Das liegt daran, dass alle diese Funktionen auf außergewöhnliche Besitzverhältnisse reagieren müssen. Und wenn Sie für diese Besitzverhältnisse in einer der Funktionen Sorge tragen müssen, dann auch in den anderen.

Sie merken schon: Jedes Mal wurde erwähnt, »was der Compiler generieren kann«. Und jedes Mal sind vor allem rohe Zeiger und dynamisch allozierte C-Arrays diejenigen Membervariablen, die besonders zu behandeln sind.[1]

17.1.2 Hilfskonstrukt per Verbot

Anstatt die nicht einfache Aufgabe anzugehen, alle diese Funktionen zu implementieren, können Sie wenigstens die Funktionen *verbieten*. Wenn Sie alle Kopier- und Verschiebeoperationen unterbinden, können auch keine Besitzverhältnisse durcheinanderkommen.

```
// https://godbolt.org/z/1Ke5W5Pj5
struct Typ {
    char* data_;              // roher Zeiger kann für unklare Besitzverhältnisse sorgen
    Typ(int n) : data_(new char[n]) {}
    ~Typ() { delete[] data_; }           // den Destruktor benötigen Sie
```

1 Es sind nicht die einzigen, aber die häufigsten und wichtigsten.

```
    Typ(const Typ&) = delete;              // keine Kopie zulassen
    Typ& operator=(const Typ&) = delete; // keine Zuweisung bitte
    Typ(Typ&&) = delete;                   // kein Verschieben
    Typ& operator=(Typ&&) = delete;        // kein Verschiebeoperator
};
```

Listing 17.1 Verbieten Sie mit »= delete« vier der großen Fünf.

Wenn Sie also »leider« einen Typ haben, der ein problematisches Feld hat (das Sie nicht loswerden können oder wollen), dann sollte zu Ihrer Sicherheit Ihr erster Schritt darin bestehen, das versehentliche Kopieren und Verschieben zu verbieten, wie in Listing 17.1 gezeigt.

Mit `= delete` hinter den kritischen Operationen verhindern Sie, dass der Compiler Ihnen ungeeignete Funktionen generiert.

17.1.3 Die Nullerregel und ihr Einsatz

Verleide ich Ihnen die rohen Zeiger? Gut! Denn des Rätsels Lösung, der Stein der Weisen, der heilige Gral ist: Nutzen Sie aus, dass der Compiler für Sie die Funktionen generieren kann. Sie müssen nur zulassen, dass er das korrekt macht.

Dies ist ein Dan-Kapitel, und eigentlich lernen Sie den Umgang mit rohen Zeigern erst in Kapitel 20, »Zeiger«. Aber da ich Ihnen die rohen Zeiger sowieso verleiden möchte, hat dieser Vorgriff seinen Vorteil. Prägen Sie sich zuerst die Alternativen ein. Sie werden in den folgenden Absätzen aber dem einen oder anderen rohen Zeiger begegnen.

Das Wichtigste dazu ist, dass Sie keine rohen Zeiger verwenden, die Daten *besitzen* – das heißt, sie mit `new` angefordert zu haben und dadurch für den Aufruf von `delete` implizit zuständig zu sein. Verwenden Sie stattdessen das, was Ihnen die Standardbibliothek bietet:

- Für große Datenmengen gibt es die Container. Verkettete Strukturen müssen Sie nicht mehr selbst aufbauen, Sie haben eine große Auswahl.
- Jedes `new` darf nur noch direkt in einen smarten Pointer wandern. Besser noch, verwenden Sie `make_shared` und `make_unique`.
- Mit den smarten Pointern benötigen Sie kein `delete` und kein `delete[]` mehr. Diese sind für Sie abgeschafft.
- Definieren Sie *keine* Operation der großen Fünf. Lassen Sie den Compiler das erledigen. Dies ist die »Nullerregel«, die »Rule of Zero«.

- Es gibt Objekte, die nicht kopiert werden *können*, zum Beispiel ein Stream oder ein Mutex. Hält Ihre Klasse solche Felder, macht der Compiler auch das Richtige: Er verbietet die Kopie. Unter Umständen bleibt die Fähigkeit zum Verschieben erhalten. Bleiben Sie hier bei der Nullerregel.

Die *C++ Core Guidelines* legen ebenfalls großen Wert auf die Kenntlichmachung von Besitz, siehe Anhang A, »Guter Code, 7. Dan: Richtlinien«. Die darin definierte *Guideline Support Library* bietet Werkzeuge wie `owner<T>` an, um einen besitzenden rohen Zeiger zu markieren. Rohe Zeiger ohne `owner<T>` besitzen ihr Ziel nicht und sind beim Kopieren wenig problematisch.

Schreiben Sie Ihre Klassen so, dass Sie die Nullerregel anwenden können

Wenn Sie statt der besitzproblematischen Felder die entsprechenden Strukturen der Standardbibliothek verwenden, generiert der Compiler Ihnen für Ihre Klasse passende Kopier- und Verschiebeoperationen sowie den Destruktor.

Definieren oder deklarieren Sie dann *keine* Operation der großen Fünf für Ihre Klasse.

Dann wird das obige Beispiel nahezu trivial. Je nach Einsatzzweck verwenden Sie einfach einen `vector` oder smarte Pointer.

```
// https://godbolt.org/z/xMbG771En
#include <vector>
#include <memory>           // unique_ptr, shared_ptr
struct Typ1 {               // automatische komplette Kopie der Ressource
    std::vector<char> data_;
    Typ1(int n) : data_(n) {}
};
struct Typ2 {               // Kopie untersagt, Verschieben möglich
    std::unique_ptr<int[]> data_;
    Typ2(int n) : data_(new int[n]) {}
};
struct Typ3 {               // Kopie erlaubt, Ressource wird dann sauber geteilt
    std::shared_ptr<Typ1> data_;
    Typ3(int n) : data_(std::make_shared<Typ1>(n)) {}
};
```

Listing 17.2 Statt eines rohen Zeigers verwenden Sie ein Standardkonstrukt und definieren keine Operation der Großen Fünf.

17.1.4 Ausnahmen von der Nullerregel

Als Erstes: Besonders Bibliotheken anderer Anbieter enthalten Dinge, die bei der Kopie unkorrekte Besitzverhältnisse hinterlassen – zum Beispiel Fenster- und Daten-

bankhandles. Oft haben jene Entwickler nicht an schädliche Kopien gedacht und sie nicht verboten. Hier empfehle ich zwei Techniken:

- Greifen Sie auf das manuelle Verbieten von Kopie und Verschieben zurück. Der Destruktor, den Sie schreiben, muss sich dann wie beim rohen Zeiger um das korrekte Entfernen kümmern.
- Wenn Sie sich mit C++ und den smarten Pointern sicher genug fühlen, dann können Sie deren *Custom Deleter* verwenden, um nicht die rohe, sondern die eingepackte Ressource zu verwenden. Der smarte Pointer regelt dann das korrekte Entfernen, und Sie können sich die großen Fünf wieder schenken. Von Fall zu Fall erzeugt der Compiler sogar dennoch brauchbare Kopier- und Verschiebeoperationen.

Als Zweites: Wenn Sie eine Klassenhierarchie schreiben, in der *virtuelle Methoden* vorkommen, dann müssen Sie den Destruktor der Basisklasse schreiben und mit `virtual` markieren (siehe Kapitel 15, »Vererbung«). Der Körper darf ruhig leer sein – und er ist es auch, wenn Sie die sonstigen Richtlinien der Nullerregel befolgt haben.

```
// https://godbolt.org/z/1xcqon7dM
struct Base {
    virtual ~Base() {}; // definieren Sie den Destruktor, machen Sie ihn virtual
    virtual void other();
};

struct Derived : public Base {
    void other() override;
};

int main() {
    Base *obj = new Derived{};
    /* ... mehr Programmzeilen hier ... */
    delete obj;       // klappt, weil Base::~Base virtual ist
}
```

Listing 17.3 In einer Hierarchie mit virtuellen Methoden müssen Sie den Destruktor der Basisklasse definieren und virtuell markieren.

Würden Sie den Destruktor nicht deklarieren, dann würde der Compiler einen generieren. Der Compiler würde diesen als nicht-`virtual` erzeugen, und das wäre schlecht: Wenn der Destruktor nicht `virtual` ist, kann beim `delete` eines Pointers der abgeleiteten Klassen auch mal der falsche Destruktor aufgerufen werden, wie dies im Beispiel in der Zeile `delete obj` in `main()` der Fall wäre. Würden Sie `virtual ~Base() {}` weglassen, wäre obiges Programm fehlerhaft.

Doch halt: Obiges Programm folgt gar nicht den Richtlinien der Nullerregel. Ich habe mit `Base *obj` einen rohen Zeiger als Besitzer verwendet. Der muss noch weg. Das können Sie zum Beispiel mithilfe eines `shared_ptr`. Die gute Nachricht ist: Wenn Sie den verwenden, haben Sie auch bei vergessenem virtuellem Destruktor kein Problem mit falschem Wegräumen. In den `shared_ptr` sind Mechanismen eingebaut, die mehr tun als der Compiler alleine: Beim Entfernen seines Schützlings ruft er immer den korrekten Destruktor auf, virtuell oder nicht.

```
int main() {
    shared_ptr<Base> obj{ new Derived{} };
    /* ... mehr Programmzeilen hier ... */
} // obj wird korrekt weggeräumt
```

Listing 17.4 shared_ptr ruft immer den richtigen Destruktor auf, virtuell oder nicht.

Das ist ein Grund mehr, keine rohen Zeiger als Besitzer zu verwenden.

Wenn Sie einen polymorphen Zeiger benötigen, verwenden Sie statt roher Zeiger den `shared_ptr`.

Sie werden wissen, wann ein Zeiger »polymorph« ist, nämlich wenn Sie Zeiger innerhalb einer Objekthierarchie erzeugen, in der die Klassen virtuelle Methoden haben: wenn dann also der Typ der Zeigervariablen (hier `Base*`) ein anderer ist, als der Typ des `new` (hier `Derived*`). Mit `unique_ptr` funktioniert das nicht ohne weitere Mechanismen.

Sie können Zuweisung und Verschiebezuweisung gemeinsam implementieren, indem Sie das Argument by Value übergeben (als eine vom Compiler erzeugte Kopie) und den Inhalt per `swap` austauschen:

```
struct Data {

    // ... andere Spezialfunktionen und Implementierung ...
    Data& operator=(Data copy) { // by Value, Zuweisung, Verschiebezuweisung
        copy.swap(*this);
        return *this;
    }

    void swap(Data& other) { // Mitgliedsfunktion
        using std::swap;
        // ... elementweiser Tausch ...
    }

    friend void swap(Data&a, Data&b); // friend: freie Funktion
}
```

Die `swap`-Funktion ist sowieso nützlich. Für die Zusammenarbeit mit Containern bieten Sie am besten auch die freie `swap`-Funktion an.

17.1.5 Nullerregel, Dreierregel, Fünferregel, Viereinhalberregel

Es gibt viele Tipps dazu, welche der großen fünf Spezialfunktionen Sie implementieren sollen. Ich fasse die verschiedenen Aspekte hier einmal zusammen:

- **Nullerregel**

  Prüfen Sie zuerst, ob es möglich ist, gar keinen der Spezialfunktionen zu implementieren. Lassen Sie diese vom Compiler generieren oder deklarieren Sie sie mit `= default` (so dokumentieren Sie, dass Sie den Aspekt nicht vergessen haben). Das ist dann anwendbar, wenn Ihre Klasse keine Ressourcenverwaltung selbst implementieren muss.

- **Viereinhalberregel**

  Wenn Ihre Klasse jedoch Ressourcen verwaltet, also angefangen mit einem Destruktor, implementieren Sie alle fünf, können aber Zuweisung und Verschiebezuweisung zusammenfassen (also vier). Nutzen Sie ein selbst implementiertes `swap()` (viereinhalb).

- **Fünferregel**

  Implementieren Sie Zuweisung und Verschiebezuweisung getrennt und bieten Sie optional `swap` an.

- **Dreierregel**

  Ihre Klasse verwaltet Ressourcen, aber der Nutzen mittels Verschieben ist nicht so relevant: Schreiben Sie Kopierkonstruktor, Zuweisungsoperator und Destruktor. Sollte der Compiler einmal verschieben wollen, kopiert er stattdessen.

17.2 RAII – Resource Acquisition Is Initialization

RAII heißt, beim Erzeugen eines Objekts – einer Ressource – auch deren *Initialisierung* zu erledigen. In C++ geschieht das im Konstruktor. Wichtig ist, beim Verlassen des Konstruktors *immer* ein *gültiges* Objekt zurückzulassen.

Das heißt insbesondere, dass auch ein Fehler kein ungültiges Objekt herumliegen lassen darf. Tritt während der Initialisierung des Objekts im Konstruktor ein Fehler auf, dürfen Sie keine Datenstrukturen halb erzeugt herumliegen lassen oder noch Ressourcen blockieren. Ressource kann hier vieles bedeuten, zum Beispiel einen Pointer auf schon reservierten Speicher, eine geöffnete Datei oder ein *Lock* für den gegenseitigen Ausschluss mehrerer Prozesse.

Im Fehlerfall sollte der Konstruktor mit einer *Exception* beendet werden, damit höhere Ebenen ebenfalls ihre Ressourcen wieder freigeben können, zum Beispiel angeforderten Speicher. Doch auch die Exception will sorgfältig ausgelöst werden: Zuvor angeforderte Ressourcen müssen vorher (oder in einem eigenen `catch` mit *rethrow*) wieder freigegeben werden, da die aktuelle Instanz nach dem `throw` (normalerweise) nicht mehr die Kontrolle zurückerhält.

Ein Aspekt von »gültig« ist, dass der zugehörige Destruktor das Objekt *in jedem Fall* komplett selbsttätig wieder entfernen können muss und alle noch bestehenden Ressourcen auch wieder freigegeben werden. Zusammengenommen, implementieren Sie so sehr effektiv »Stack-based Resource Management«.

17.2.1 Ein Beispiel mit C

Eine typische C-Programmierschnittstelle (Application Programming Interface – API) verlagert Verwaltungsaufgaben häufig in den Programmcode, der die API verwendet. Ein Dateiobjekt oder eine Datenbankverbindung muss geöffnet und mit einem symmetrischen Aufruf wieder geschlossen werden. *Handles* zu Schriftarten, Pinseln oder Audio-Video-Codecs werden geholt und explizit wieder freigegeben. Wird der freigebende Aufruf nicht durchgeführt – sei es durch ein unvorhergesehenes Ereignis oder durch schlichtes Vergessen –, werden die damit verbundenen Ressourcen nicht wieder freigegeben. Die Datei bleibt geöffnet, die Datenbankverbindung wird nicht geschlossen, die Handles auf Pinsel und Schriften gehen aus.

Dazu passend wird in der C-Welt häufig mit speziellen Rückgabewerten gearbeitet, die einen Fehler darstellen sollen. Als Zeichen dafür, dass der Client die Verwendung der Ressource unterlassen soll, nutzen viele Funktionen bestimmte Fehlercodes. Zum Beispiel liefert `mktime` statt der angeforderten Konvertierung im Fehlerfall eine –1 zurück. Und den Rückgabewert von `malloc` müssen Sie gegen den *Nullpointer* prüfen.

Üblicherweise ist der Fehlerfall bei der Prüfung das *selten erwartete* Ergebnis. Fast könnte man die Prüfung weglassen, weil dieser Fall »sowieso nicht auftritt«.[2] Dann kann man nur hoffen, dass der Code nicht irgendwann einmal in sicherheitskritischem Umfeld eingesetzt wird.

Und da es *selten erwartet* wird, wären wir bei der *Ausnahme*: In der C++-Welt stehen für solche Fehler Exceptions zur Verfügung. Der Rückgabewert braucht nicht überprüft zu werden, da ein Konstruktor ein Objekt nur in einem *gültigen Zustand* hinterlassen darf. Wird der Konstruktor per Exception verlassen, wird der vorbelegte Speicher vom Compiler wieder freigegeben. Passiert die Exception dabei mehrere Konstruktoren im *Stack* (der Liste der aktuellen Funktionsaufrufe) aufwärts, ohne per passendem `catch` gefangen zu werden, dann werden alle bisherigen Konstruktionen

2 Achtung: Ironie!

rückgängig gemacht. Nur *genau dann*, wenn ein Konstruktor komplett durchlaufen wurde, gilt das Objekt als gültig und vollständig erzeugt.

Eine kleine schlanke Beispielklasse, die nur einen Zeichenpuffer kapseln soll, könnte so aussehen:

```
// https://godbolt.org/z/9eeG575Wz
struct Puffer {
  const char *data;
  explicit Puffer(unsigned sz): data(new char[sz]) {}
  ~Puffer() { delete[] data; }
  Puffer(const Puffer&) = delete;
  Puffer& operator=(const Puffer&) = delete;
};
```

Listing 17.5 Eine RAII-Zeichenpufferklasse

Es sei erwähnt, dass – ganz nach dem Motto »Sixteen ways to stack a cat« – die gleiche Aufgabe besser durch `unique_ptr<char[]>` zu erledigen wäre; dies soll nur ein einfaches Beispiel sein.

In C stünde hier meist ein `malloc`, und (sorgfältige oder paranoide[3]) Programmierer müssten den Rückgabewert darauf prüfen, ob der Speicher korrekt alloziert wurde. Wenn nicht, muss der Vorgang irgendwie abgebrochen werden. In C++ liefert `new` immer einen gültigen *Pointer* oder wirft eine Exception `bad_alloc`.[4] Es gibt also zwei mögliche Ausgänge aus dem Konstruktor:

- **normaler Durchlauf**

  Der Speicher für `data` wurde angefordert, und der Pointer ist gültig. Das `Puffer`-Objekt gilt als korrekt erzeugt, und wenn es weggeräumt werden soll, erledigt der Destruktor dies vorbildlich.

- **Exception `bad_alloc`**

  Der Speicher wurde nicht alloziert, die Exception verlässt den Konstruktor. Damit gilt der `Puffer` als nicht erzeugt, und es werden auch keine belegten Ressourcen zurückgelassen. Weil das Objekt nicht erzeugt wurde, wird der Destruktor *in keinem Fall* aufgerufen. Auch wenn `data` also uninitialisiert ist und damit ein `delete[]` äußerst gefährlich wäre – es tritt niemals auf.

Die mit `= delete` gelöschten Methoden sorgen dafür, dass der Puffer nicht aus Versehen kopiert wird und dann die interne Datenstruktur mehrfach freigegeben wird: ein weiterer Grund, besser `unique_ptr` zu verwenden.

3 Was oft das Gleiche ist.

4 Viele Compiler können angewiesen werden, ganz ohne Exceptions zu arbeiten. Laut Standard ist das korrektes Verhalten. Dies kommt vor allem im Embedded- und Realtime-Bereich vor.

17.2.2 Besitzende Raw-Pointer

In komplexen Objekthierarchien dürfen Sie nicht vergessen, dass Sie auch darauf vorbereitet sein müssen, dass die Initialisierung von Objektvariablen (*Membervariablen*) mit einer Exception fehlschlagen kann. Auch wenn Sie sie nicht per `catch` behandeln und verschlucken, sondern die Ausnahme eigentlich durchlassen möchten, müssen Sie gegebenenfalls zuvor gemachte Initialisierungen rückgängig machen, bevor die Exception den aktuellen Konstruktor verlässt.

Anders als bei der obigen `Puffer`-Klasse sieht es bei `StereoImage` nicht so einfach aus:

```
// https://godbolt.org/z/TK3vTv1EM
struct StereoImage {
  Image *left, *right;
  StereoImage()
  : left(new Image)
  , right(new Image) // Gefahr!
  { }
  ~StereoImage() {
      delete left;
      delete right;
  }
};
```

Listing 17.6 Wirft »right« eine Exception, entsteht mit »left« ein Leck.

Schlüge hier die Erzeugung von `right` mit einer Exception fehl, würde der Speicher, der für `left` schon alloziert wurde, nicht freigegeben. Da der Konstruktor mit einer Exception verlassen wurde, gilt das `StereoImage` als nicht erzeugt, und dessen Destruktor wird nicht aufgerufen werden. Wohl würde der Compiler die erfolgreiche Initialisierung von `left` rückgängig machen können, indem dessen Destruktor aufgerufen wird, doch ist dies hier ein *Raw-Pointer* ohne Destruktor! Ein anderer Mechanismus muss her – ein `catch` mit einem *rethrow* oder etwas anderes als ein Raw-Pointer für die Felder von `StereoImage`.

Hier ein Beispiel (unter vielen), wie Sie diese Schwachstelle umgehen können:

```
// https://godbolt.org/z/cs4eEEGrf
#include <memory> // unique_ptr
struct Image {
    /* ... */
};
struct StereoImage {
  std::unique_ptr<Image> left, right;
  StereoImage()
```

```
  : left(new Image)
  , right(new Image)
  { }
};
```

Listing 17.7 Korrektes RAII für »StereoImage«

Schlägt nun die Initialisierung von `right` mit einer Exception fehl, gilt `StereoImage` – wie bisher – als nicht erzeugt. Der Destruktor von `StereoImage` wird nicht aufgerufen. Nun hat `left` aber einen vollwertigen Destruktor, und der Compiler kann diesen bei der Rückabwicklung aufrufen. Und der wiederum gibt den Speicher von `left` frei.

Der Raw-Pointer belegt hier die Ressource »Speicher«. Gesagtes gilt aber für alle Arten von Ressourcen ebenso: *Dateihandles*, *Sockets*, *Mutexe* und *Semaphoren* etc.

17.2.3 Von C nach C++

Wie zu Beginn des Kapitels bemerkt, werden C++-Entwickler auch häufig C-APIs verwenden. Typisch ist das paarweise *Anfordern* und *Freigeben* einer Ressource über jene API. Verwenden Sie zum Beispiel das serverlose Datenbankinterface *Sqlite*, könnte ein Codefragment vereinfacht so aussehen:

```
// https://godbolt.org/z/GK4W1nKd4
#include <string>
#include <stdexcept>
#include <sqlite3.h>
using std::string; using std::runtime_error;

void dbExec(const string &dbname, const string &sql) {
  sqlite3 *db;
  int errCode = sqlite3_open(dbname.c_str(), &db);  // Acquire
  if(errCode) {
    throw runtime_error("Fehler beim Öffnen der DB.");
  }
  errCode = sqlite3_exec(db, sql.c_str(), nullptr, nullptr, nullptr);
  if(errCode) {
    throw runtime_error("Fehler SQL-Exec.");        //↯ Nicht gut!
  }
  errCode = sqlite3_close(db);                      // Release
}
```

Listing 17.8 C-API in C++-Code

Hier wird mit `sqlite3_open` die Ressource *Datenbank* geholt, mit `sqlite3_close` wieder freigegeben und dazwischen mit `sqlite3_exec` auf ihr operiert. Da ist es natürlich keine gute Idee, die Funktion mit einer Exception zu verlassen:

- Die erste Exception ist korrekt, denn wenn `sqlite3_open` nicht erfolgreich war, muss auch `sqlite3_close` nicht aufgerufen werden.[5]
- Die zweite Exception verhindert jedoch, dass `sqlite3_close` aufgerufen wird, und die Ressource würde blockiert – oder zumindest nicht freigegeben –, was je nach Art der Ressource unterschiedliche Auswirkungen haben kann.

Man könnte natürlich die gesamte API von Sqlite3 in eine C++-API umwandeln, aber das ist sicherlich ein großes Unterfangen. Abgesehen davon hat diese Arbeit für eine populäre API wie *Sqlite3* sicher schon jemand gemacht – der wird dann hoffentlich auch die Pflege übernehmen, wenn sich in der C-API etwas ändert.

Zu Recht möchte man sich aber nicht eine weitere Abhängigkeit einhandeln. Eine kleinere Lösung tut es vielleicht auch. Eine einfache Standardmethode, solchen Code in RAII-Code zu transformieren, besteht darin, die Ressource in eine Wrapper-Klasse »einzuwickeln«:

```
// https://godbolt.org/z/hqrPjGjnK
#include <sqlite3.h>
class DbWrapper {
  sqlite3 *db_;
public:
  // acquire resource
  DbWrapper(const string& dbname)
    : db_{nullptr}
  {
    const int errCode = sqlite3_open(dbname.c_str(), &db_);
    if(errCode)
      throw runtime_error("Fehler beim Öffnen"); // verhindert sqlite3_close
  }

  // release resource
  ~DbWrapper() {
    sqlite3_close(db_);  // Release
  }
  // access Resource
  sqlite3* operator*() { return db_; }
```

5 Um genau zu sein, steht in der Dokumentation, dass man es »sollte« – aber nicht »muss«. Wir nehmen das Beispiel exemplarisch für den häufigen Fall bei der Ressourcenverwaltung, bei der nur dann weggeräumt werden soll, wenn das Anfordern erfolgreich war.

```
  // Keine Kopie und Zuweisung
  DbWrapper(const DbWrapper&) = delete;
  DbWrapper& operator=(const DbWrapper&) = delete;
};
void dbExec(const string &dbname, const string &sql) {
  DbWrapper db { dbname };
  const int errCode = sqlite3_exec(*db, sql.c_str(), nullptr,
    nullptr, nullptr);
  if(errCode)
    throw runtime_error("Fehler SQL-Exec."); // Jetzt geht es!
}
```

Listing 17.9 C-API mit einfachem RAII

Mit diesem Wrapper kann ohne Probleme die Exception nach `sqlite3_exec` geworfen werden. Beim Verlassen des Gültigkeitsbereichs von `db` – per Exception oder normal – wird der Destruktor aufgerufen und damit `sqlite3_close`.

Bei einer Exception im Konstruktor gilt das Objekt als nicht erzeugt, und deswegen wird der Destruktor mit `sqlite3_close` nicht aufgerufen. Eine gute, einfache RAII-Lösung.

Mit den beiden Zeilen mit `= delete` verhindern wir, dass versehentlich Zuweisungen und Kopien angelegt werden, die eine mehrfache Freigabe der gleichen Ressource `db_` verursachen würden.

17.2.4 Es muss nicht immer eine Exception sein

Auch wenn ich mich bei der Besprechung von RAII in diesem Kapitel hauptsächlich um Exceptions kümmere, ist RAII nicht zwangsläufig mit dem Auslösen von Exceptions verknüpft. Sie müssen sich nur die wichtigen Prinzipien in Erinnerung rufen, und die Fußangeln sind besonders beim Umgang mit Exceptions vorhanden.

Es ist durchaus möglich, RAII ohne `throw` zu implementieren und stattdessen Benutzer nach dem Konstruieren den Erfolg der Initialisierung prüfen zu lassen – zum Beispiel mit einem *Cast* nach `bool`:

```
// https://godbolt.org/z/5bW9Evorc
#include <string>
#include <sqlite3.h>
class DbWrapper {
  sqlite3 *db_;
public:
  DbWrapper(const std::string& dbname)
    : db_{nullptr}
```

```
  {
    const int errCode = sqlite3_open(dbname.c_str(), &db_);
    if(errCode) db_ = nullptr;  // als 'nicht erfolgreich' markieren
  }
  explicit operator bool() const {
    return db_ != nullptr;      // Markierung auswerten
  }
  ~DbWrapper() {
    if(db_) sqlite3_close(db_);
  }
  // ... Rest wie zuvor ...
};
bool dbExec(const std::string &dbname, const std::string &sql) {
  DbWrapper db { dbname };
  if(db) {                  // prüfe auf erfolgreiche Initialisierung
    const int errCode = sqlite3_exec(*db,sql.c_str(),nullptr,nullptr,nullptr);
    if(errCode)
      return false; // immer noch korrektes RAII
  }
  return (bool)db;
}
```

Listing 17.10 C-API mit einfachem RAII, ohne throw

Entscheidend ist, dass Sie alle Tätigkeiten des Konstruktors im Destruktor wieder rückgängig machen können – egal, ob die Initialisierung erfolgreich war oder nicht. Das wird hier durch `db_ = nullptr` und den Check im Destruktor sichergestellt. Und mithilfe von `operator bool()` lässt sich auch von außen prüfen, ob die Initialisierung Erfolg hatte. Im Beispiel ist dieser zusätzlich mit `explicit` markiert. In der Standardbibliothek funktionieren zum Beispiel die *Streams* auf diese Weise, doch dazu gleich mehr.

17.2.5 Mehrere Konstruktoren

Es ist auch darauf zu achten, dass alle Wege, das Objekt zu erzeugen, eine gültige Instanz hinterlassen müssen. In der Standardbibliothek ist `ostream` ein Beispiel dafür:

- `ostream os` erzeugt einen Datenstrom zu einer noch nicht festgelegten Datei,
- `ostream os {"file.txt"}` öffnet die Datei, wenn möglich.

Auf jeden Fall kann der Destruktor den Stream wieder wegräumen. Operationen, zum Beispiel via `<<`, landen bei der zweiten Variante im Nirwana, wenn es beim Öffnen der Datei einen Fehler gab. Daher ist vor der Verwendung per `if(!os)` schnell eben zu

prüfen, ob der Stream wirklich fürs Schreiben bereit ist. Des Pudels Kern ist hier jedoch, dass `os` in jedem Fall zugewiesen, nachträglich geöffnet und vor allem wieder korrekt weggeräumt werden kann. Als alternative Designentscheidung hätte hier in der Variante `ostream os{"file.txt"}` im Fehlerfall eine Exception geworfen werden können. So müssen Sie das *C-Pattern* verwenden: zunächst den Rückgabewert prüfen. Das wird auch prompt häufig vergessen und führt dann zu Überraschungen beim nächsten `<<`.

17.2.6 Mehrphasige Initialisierung

Manchmal empfiehlt sich eine noch stärkere Abweichung von der reinen RAII-Lehre. Häufig ist es besser, ein Objekt in mehreren Phasen zu initialisieren. Typischerweise sind Fenster-APIs so zu benutzen, dass im ersten Schritt – im Konstruktor – die nicht sichtbaren Dinge erzeugt werden, und dann in einer `init()`-Methode die tatsächliche Visualisierung stattfindet. Zwischen diesen beiden Phasen werden Fenster und Komponenten miteinander verknüpft, im Layoutmanager eingebettet, und häufig werden weitere dynamische Aufgaben erledigt.

Aber egal, ob nur der Konstruktor und noch *kein* `init` oder Konstruktor *und* `init` ausgeführt wurden: Der Destruktor muss das Objekt *immer* vollständig wegräumen können.

17.2.7 Definieren, wo es gebraucht wird

Beinahe direkt aus RAII folgt, dass man seine Variablen »so spät wie möglich und so früh wie nötig« definiert – also nah an ihrer tatsächlichen Verwendung.

Ausnahmen sind hier natürlich enge Schleifen, aus denen man kostspielige Konstruktoraufrufe durch die Verwendung eines temporären Objekts ersetzen möchte. Doch sollte man an dieser Stelle Compiler und Standardbibliothek nicht unterschätzen – ein kurzer `string` in einer »engen« Schleife ist vielleicht *gerade noch* unerwünscht, aber in einer größeren kann man den `string` gut dort definieren, wo er gebraucht wird. Elementare Datentypen, wie `int` und dergleichen, wird der Compiler aber ohnehin wegoptimieren.

17.2.8 Nothrow-new

Es soll nicht verschwiegen werden, dass es in manchen Fällen durchaus sinnvoll und erwünscht sein kann, *keine* `bad_alloc`-Exception bedenken zu müssen. Zum Beispiel gibt es Umgebungen und Compiler, die ganz ohne Exceptions auskommen müssen. Dies ist im Embedded- und Realtime-Bereich verbreitet, auch wenn diese Sonderbehandlung dort in den vergangenen Jahren abgenommen hat. Und manchmal ist man

sich ja durchaus *sicher*, dass hier keine Exception geworfen werden kann, zum Beispiel weil man seine eigene Speicherverwaltung geschrieben hat oder genau weiß, dass die Speicheranforderung nicht fehlschlägt.

Zu diesem Zweck können Sie mit einer besonderen Form des `new` das `nullptr`-Verhalten erzwingen:

```
// https://godbolt.org/z/s8KoE4ETs
#include <new> // nothrow
std::string *ps = new(std::nothrow) std::string{};
if(ps == nullptr) {
   std::cerr << "Die Speicheranforderung ging schief\n";
   return SOME_ERROR;
}
```

Listing 17.11 Nothrow-new wirft kein »bad_alloc«, sondern liefert »nullptr« zurück.

Ein `new`, das auf diese Weise aufgerufen wurde, wirft niemals eine Exception (der aufgerufene Konstruktor eventuell schon). Stattdessen liefert es im Fehlerfall einen `nullptr` zurück.

Tipp

Präferieren Sie RAII beim Design Ihrer Objekte; schreiben Sie Programmstücke, die RAII nutzen.