# Programmieren mit modernem C++

eine „kurze“ Einführung

René Richter

Version: 2019-04-14

*C++ wird verbreitet in Lehre und Forschung eingesetzt. Dies hat manche überrascht, die – zu Recht – darauf hinwiesen, dass C++ weder die kleinste noch die reinste Sprache ist, die jemals entwickelt wurde. Aber C++ ist*

> *klar genug, um Basiskonzepte erfolgreich zu lehren,*
> *praxisnah, effizient und flexibel genug für anspruchsvolle Projekte,*
> *verfügbar genug für Organisationen und Projektgruppen,*
> *umfassend genug, um fortgeschrittene Konzepte und Techniken zu lehren und*
> *kommerziell genug, um das Gelernte in nicht akademischen Umgebungen zu nutzen.*

*C++ ist eine Sprache, mit der man wachsen kann.*

– Bjarne Stroustrup

*If you can look at a screenful of code and tell that it's C++17, then we've changed the way we program C++.*

– Herb Sutter

Einsatz als Schulungsmaterial „Gelbes Heft“:
1994–2001 Schülerrechenzentrum Dresden, https://srz.tu-dresden.de
2000–2012 Bildungszentrum für informationsverarbeitende Berufe Dresden /
b.i.b. International College Dresden
2005–2011 Fachhochschule der Wirtschaft Dresden

4. überarbeitete Fassung, Dresden 2019.

(c) Alle Rechte vorbehalten: René Richter, https://namespace-cpp.de

# Inhaltsverzeichnis

**A Allgemeine Bemerkungen** **1**
A.1 Sprichst du modernes C++? . . . 1
A.2 Hallo, Welt . . . 3

**B Bücher zu C++** **4**
B.1 Referenzen . . . 4
B.1.1 Standards . . . 4
B.1.2 Sprachbeschreibungen . . . 4
B.2 Lehrbücher . . . 5
B.2.1 Anfänger . . . 5
B.2.2 Fortgeschrittene . . . 5

**C C++ Sprachbeschreibung** **6**
C.1 Lexikalische und syntaktische Grundbegriffe . . . 6
C.1.1 Zeichen . . . 6
C.1.2 Symbole . . . 6
C.1.3 Weitere syntaktische Einheiten . . . 8
C.2 Datentypen, Variablen, Operationen . . . 9
C.2.1 Elementare Datentypen . . . 9
C.2.2 Variablen . . . 11
C.2.3 Konstanten und flatterhafte Variablen . . . 13
C.2.4 Operationen . . . 15
C.2.5 Auswerteregeln . . . 19
C.3 Gruppierung von Daten . . . 20
C.3.1 Felder . . . 20
C.3.2 Strukturen . . . 21
C.3.3 Erweiterte Typen . . . 21
C.3.4 Bitfelder und Vereinigungen . . . 22
C.4 Zeiger . . . 23
C.4.1 Deklaration und Initialisierung . . . 23
C.4.2 Fehlerquellen (wild pointer) . . . 23
C.4.3 Zeigerarithmetik . . . 24
C.4.4 Besondere Zeiger . . . 25
C.5 Steueranweisungen . . . 26
C.5.1 Schleifen . . . 26
C.5.2 Verzweigungen . . . 28
C.5.3 Sprunganweisungen . . . 29
C.6 Funktionen . . . 30
C.6.1 Programm-Grundbausteine . . . 30
C.6.2 Vereinbarung . . . 30
C.6.3 Parameter . . . 32
C.6.4 Hauptprogramm . . . 33
C.6.5 Deklarationsdateien . . . 33

C.7 Vorverarbeitung . . . 34
C.7.1 Makrokonstanten und -funktionen . . . 34
C.7.2 Fehler und Gefahren . . . 35
C.7.3 Zeichenkettenmanipulation . . . 36
C.7.4 Vordefinierte Makros . . . 36
C.7.5 Einlesen von Dateien . . . 37
C.7.6 Bedingte Übersetzung . . . 37
C.8 Programmorganisation . . . 38
C.8.1 Übersetzungsprozess . . . 38
C.8.2 Modularisierung . . . 38
C.8.3 Namensräume . . . 39
C.9 Klassenkonzept . . . 42
C.9.1 Objektbasierte Programmierung . . . 42
C.9.2 Generische Programmierung . . . 45
C.9.3 Überladen von Operatoren . . . 46
C.9.4 Vererbung . . . 50
C.10 Ausnahmebehandlung . . . 56
C.10.1 Ausnahmen werfen . . . 56
C.10.2 Ausnahmen oder Ausnahmefreiheit erklären . . . 56
C.10.3 Ausnahmen abfangen . . . 56
C.11 Freispeicherverwaltung . . . 58
C.11.1 Container und smarte Zeiger . . . 58
C.11.2 Hinter den Kulissen . . . 60
C.12 Zeichenketten . . . 62
C.12.1 C-Zeichenketten . . . 62
C.12.2 Die Klasse `string` . . . 62
C.12.3 Lokalisierung und Zeichenarten . . . 70
C.12.4 Reguläre Ausdrücke . . . 71
C.13 Ein- und Ausgabe . . . 74
C.13.1 Datenströme . . . 74
C.13.2 Formatierung . . . 78
C.14 Datencontainer . . . 82
C.14.1 Verarbeitung von Datenfolgen und Datenhaltung . . . 82
C.14.2 Containerarten . . . 84
C.14.3 Allgemeine Containereigenschaften . . . 86
C.14.4 Spezielle Containeroperationen . . . 87
C.14.5 Kosten von Operationen . . . 88
C.14.6 Container-Adapter . . . 89
C.14.7 Iteratoren . . . 90
C.14.8 Iterator-Adapter . . . 91
C.15 Algorithmen . . . 92
C.15.1 Überblick . . . 92
C.15.2 Nichtmodifizierende Algorithmen . . . 94
C.15.3 Modifizierende Algorithmen . . . 96
C.15.4 Mutierende Algorithmen . . . 98

C.15.5 Sortierende Algorithmen . . . 99
C.15.6 Numerische Algorithmen . . . 102
C.16 Hilfsfunktionen . . . 104
C.16.1 Funktoren . . . 104
C.16.2 Funktorenbibliothek . . . 106
C.17 Mathematik . . . 108
C.17.1 Mathematische Funktionen . . . 108
C.17.2 Komplexe Zahlen . . . 108
C.17.3 Tupel . . . 110
C.17.4 Verhältnissse . . . 110
C.17.5 Zufallszahlen . . . 111
C.17.6 Bitmengen . . . 114
C.17.7 Wertfelder . . . 114
C.18 Zeit . . . 116
C.18.1 Abstraktion vom Rechnertakt . . . 116
C.18.2 Rückgriff auf C-Bibliothek . . . 117
C.19 Datenkapseln . . . 118
C.19.1 `std::optional<T>` . . . 118
C.19.2 `std::any` . . . 118
C.19.3 `std::variant<T1,...,Tn>` . . . 119
C.20 Dateisystem . . . 121
C.20.1 Pfade . . . 121
C.20.2 Verzeichnisse und Dateien anlegen, kopieren, umbenennen, löschen 123
C.21 Parallelverarbeitung . . . 124
C.21.1 Leichtgewichtige Prozesse . . . 124
C.21.2 Gemeinsam genutzte Ressourcen . . . 125
C.21.3 Verzögerte Auswertung . . . 128
C.21.4 Sperrenfreie Kommunikation . . . 129

**D Differenzen** **130**
D.1 Sprachbarrieren . . . 130
D.1.1 Plattformabhängigkeiten . . . 130
D.1.2 C++ und C . . . 130
D.1.3 Einbinden von Modulen aus C und anderen Sprachen . . . 131
D.1.4 C++ von 1998 nach 2011 und später . . . 131
D.2 Geschmacksfragen . . . 132
D.2.1 Stil . . . 132
D.2.2 Empfehlungen . . . 132

*C++ ist das C, das wir machen wollten, jedoch nicht konnten.*
– Larry Rosler, ANSI C-Komitee

*Kenner der Semantik würden ++C gegenüber C++ vorziehen.*
*Die Sprache wurde auch nicht D genannt, denn sie ist eine Erweiterung von C*
*und versucht nicht, Probleme dadurch zu lösen, dass Funktionen weggelassen werden.*
– Bjarne Stroustrup

# A Allgemeine Bemerkungen

## A.1 Sprichst du modernes C++?

Die Programmiersprache C++ entwickelt sich kontinuierlich weiter. Dabei blickt sie auf eine lange Entwicklungsgeschichte zurück (Tab. 1). Hervorgegangen aus C, der Systemprogrammiersprache des Betriebssystems UNIX, wurde sie durch Anregungen aus anderen Sprachen (Simula, BCPL) stark beeinflusst.

C kombinierte Hochsprachenelemente mit der Effizienz von Assembler, enthielt aber keine Abhängigkeit von Hardware und Betriebssystem. Systemabhängige Routinen, in Bibliotheken gekapselt, wurden bei Bedarf eingebunden. Das erlaubte portablen Quelltext. C verdrängte Assembler fast völlig unter UNIX, dessen Kern zu 98% in C geschrieben wurde. C++ lässt sich zu einem gewissen Grad als Obermenge von C auffassen (Tab. 2) und mit in C und anderen Sprachen geschriebenen Programmteilen verbinden. Kleine Verbesserungen („a better C“) wirken sich schon auf prozedural geschriebene Programme positiv aus.

Der Schwerpunkt von C++ liegt auf Abstraktion und Erweiterbarkeit, ohne dabei die Effizienz von C aufzugeben. So können Sprachmittel geschaffen werden, die näher an der zu lösenden Aufgabe liegen. C++ kombiniert mehrere Programmierstile (Multi-Paradigmen-Sprache), ist keine „reine“ Sprache, sondern an praktischen Erfordernissen des Programmieralltags orientiert.

Ein *Klassenkonzept* unterstützt Modularisierung und Geheimnisprinzip. Trennung von Schnittstelle und Implementierung, strengere Typprüfung und Kapselung erleichtern

Tabelle 1: Entwicklung von C++.

| | | | |
|---|---|---|---|
| 1953 | FORTRAN | 1980 | C with classes |
| 1960 | ALGOL | 1983 | C++ |
| 1967 | Simula, BCPL | 1990 | Annotated Reference Manual (ARM) |
| 1970 | Unix, B | | Komitees ANSI X3J16/ISO WG21 |
| 1973 | K&R C ↗ | 1998 | Standard ISO 14882:1998 (C++98) |
| 1983 | ANSI C | 2005 | Technical Report TR1, Beginn „C++0x“ |
| 1989 | C ISO 9899:1989 (C89) | 2011 | ISO 14882:2011 (C++11) |
| 1999 | C ISO 9899:1999 (C99) | 2014 | ISO 14882:2014 (C++14) |
| 2011 | C ISO 9899:2011 (C11) | 2017 | ISO 14882:2017 (C++17) |

Tabelle 2: Schlüsselwörter, nach Aufgaben geordnet (aus C **fett**, C++11 *kursiv*).

| | |
|---|---|
| Grunddatentypen | |
| – Wahrheitswerte | `bool false true` |
| – Zeichen | **char** *char16_t char32_t* `wchar_t` |
| – Zahlen | **double float int** |
| – Platzbedarf, Vorzeichen | **long short signed unsigned** |
| – weitere | *auto* **enum typedef void** |
| – Modifizierer | *alignas* **const** *constexpr* `mutable` **volatile** |
| – Lebensdauer/Ort | **extern static** *thread_local* |
| Zusammengesetzte Typen | `class` **struct union** |
| – Klassen | `explicit this virtual` |
| – Zugriffsrechte | `friend private protected public` |
| Typinformation | *alignof auto decltype* **sizeof** `typeid typename` |
| Typumwandlung | `const_cast dynamic_cast reinterpret_cast static_cast` |
| Ablaufsteuerung | |
| – Verzweigung/Schleife | **do else for if switch while** |
| – Sprung | **break case continue default goto** |
| – Fehlerbehandlung | `catch` *noexcept static_assert* `throw try` |
| Assembler | `asm` |
| Freispeicher | `delete new` *nullptr* |
| Funktionen | `inline operator` **return** |
| Operatornamen | `and bitand bitor compl not or xor and_eq not_eq or_eq xor_eq` |
| Namensbereiche | `namespace using` |
| Schablonen | `template` |
| reserviert | *export* **register** |

Programmentwurf und -test. Typunabhängiges *generisches* Programmieren verwendet Quelltext wieder (*Schablonen*, statische Polymorphie). Dynamische Polymorphie erlaubt komplexes Verhalten verwandter Datentypen (*Vererbung*) zur Laufzeit.

Die Standardbibliothek stellt *Container* (Zeichenketten, dynamische Felder, Listen, Tabellen, Stapel, Warteschlangen) zur Datenorganisation, typische *Algorithmen* (Suchen, Sortieren, Ersetzen), Mittel zu Internationalisierung und Ein-/Ausgabe bereit.

Überarbeitungen des Standards („modernes“ C++11/C++14/C++17) räumen mit Ungereimtheiten auf, fügen neue Sprachelemente und Bibliotheken (z.B. für *nebenläufige Programmierung*) hinzu. Freispeicherverwaltung wird durch „intelligente“ Zeiger automatisiert. In vielen Merkmalen fühlt sich modernes C++ wie eine neue Sprache an, nicht mehr als Aufsatz zu C. Mittlerweile arbeitet das Komitee WG21 an C++2a.

## A.2 Hallo, Welt

Das kleinste Programm passt auf eine Zeile und macht gar nichts:

```
int main() {}
```

Innerhalb der geschweiften Klammern stehen *Anweisungen*, die das Programm ausführen soll, z.B. zum Ausgeben eines Textes. Teile von Bibliotheken (`std::cout`) müssen vor ihrer Nutzung erklärt (*deklariert*) werden, z.B. durch Einbinden (`#include <...>`) von *Header*-Dateien:

```
#include <iostream>

int main()
{
  std::cout << "Hallo, Welt\n";
  return 0; // kann weggelassen werden
}
```

Das *Hauptprogramm* `main()` meldet mit `return` einen ganzzahligen (`int`) Wert an das aufrufende Programm oder Betriebssystem zurück (`0` steht für Erfolg). Das Programm

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  std::map<std::string, int> table;
  for (std::string s; std::cin >> s;) ++table[s];

  for (const auto& [word, frequency] : table)          // C++17
    std::cout << frequency << '\t' << word << '\n';
}
```

erstellt beim Aufruf auf der Konsole

```
wordcnt < input > output
```

eine sortierte Tabelle `output` mit der Häufigkeit aller Wörter[1] aus `input`. Zeichen wie `:: >> ++ & <<` wirken zuerst befremdlich. C++ gilt zu Unrecht als *schwer* erlernbar, allerdings gibt es *viel* zu lernen. Selbst der Name C++ ist ein Insiderwitz mit einem wahren Kern: C incremented.

[1] „Wörter“ sind hier alle Zeichenfolgen ohne Leerraum.

# B Bücher zu C++

## B.1 Referenzen

### B.1.1 Standards

[**C++11**] ISO: *Information Technology - Programming Languages - C++*. International Standard ISO/IEC 14882. 3rd edn. ISO, Geneva (CH) (2011-09-01).

Der Entwurf ist bei der ISO Working Group 21 (WG21) kostenfrei einsehbar: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2011/n3242.pdf.

[**C++14**] Ebda. 4th edn. (2014-12). Entwurf: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3797.pdf.

[**C++17**] Ebda. 5th edn. (2017-12). Entwurf: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2017/n4659.pdf.

[**C99**] ISO: *Information Technology - Programming Languages - C*. International Standard ISO/IEC 9899. ISO, Geneva (CH) (1999-12-01).

C++11 setzt auf den C-Standard von 1999 auf, C++17 auf C 2011. Eine Fassung mit Korrekturen wurde von der ISO-Arbeitsgruppe veröffentlicht: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf (2007-09-07).

Der *Draft International Standard* für C11 liegt unter http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf (2011-04-12).

### B.1.2 Sprachbeschreibungen

[**TC++PL3**] Bjarne Stroustrup: *Die C++ Programmiersprache*. 3. Auflage. Addison-Wesley (1998).

Dicht und informativ. In Englisch: *The C++ Programming Language*.

[**D&E**] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley (1994).

Rückblick des Entwicklers, welche Entscheidungen C++ zu dem machten, was es heute ist. Sehr lesenswert, auch in deutsch (1994).

[**ARM**] Margareth A. Ellis, Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley (1990).

Kommentiert und begründet Sprachkonstruktionen und Compilerregeln. Vorlage für den ersten Standardentwurf.

[**Josuttis**] Nicolai Josuttis: *Die C++ Standardbibliothek*. Addison-Wesley (1996).

Beschreibt ein Zwischenstadium des Standardisierungsprozesses, vor allem die Standard Template Library (STL).

[**K&R**] Brian W. Kernighan, Dennis M. Ritchie: *The C Progamming Language*. 2nd edn. Prentice-Hall 1988.

Beschreibung und Lehrbuch zu ANSI C.

## B.2 Lehrbücher

### B.2.1 Anfänger

[**Koenig&Moo**] Andrew Koenig, Barbara Moo: *Intensivkurs C++.* Pearson (2003).

Für Anfänger ohne Programmiervorkenntnisse, beschränkt auf wichtige Sprachmerkmale und Bibliotheken.

[**Intro**] Bjarne Stroustrup: *Einführung in die Programmierung mit C++.* Addison-Wesley/Pearson (2009).

Lehrt allgemeine Programmierkonzepte und untersetzt dies mit modernem C++03. Engl. Programming Principles and Practice using C++.

[**TiC++**] Bruce Eckel: *In C++ denken.* 2. Aufl. Markt & Technik (1998).

Betont den objektorientierten Ansatz. Nicht ganz frisch, aber als Einstieg in OOP geeignet. Engl. *Thinking in C++*: http://eckelbooks.starlinger.org/.

[**Kaiser**] Ulrich Kaiser: *C/C++.* Galileo (2000).

Beschreibt zuerst C, dann C++, enthält aber viele gute Übungsaufgaben und Beispiele.

### B.2.2 Fortgeschrittene

[**Ruminations**] Andrew Koenig, Barbara Moo: *Ruminations on C++.* Addison-Wesley (1997).

Einblicke in den Entwurf objektorientierter und generischer Programme.

[**Effective C++**] Scott Meyers: *Effective C++.* 3rd edn, Addison-Wesley (2005).

55 ways to improve your programs...

[**Exceptional C++**] Herb Sutter: *Exceptional C++.* Addison-Wesley (2000).

Ausnahmebehandlung richtig gemacht: Beiträge aus Guru of the Week.

[**Alexandrescu**] Andrei Alexandrescu: *Modern C++ Design.* Addison-Wesley (2001).

Anspruchsvoll: Schablonen ausgereizt. Auch in deutsch.

[**SciEng**] John J. Barton, Lee R. Nackman: *Scientific and Engineering C++: An Introduction with advanced techniques and examples.* Addison-Wesley (1994).

Objektorientierte und generische Programmierung, Umsetzung algebraischer Strukturen in C++, Einkapselung von Legacy-Systemen (FORTRAN).

[**CLR**] Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein: *Algorithmen. Eine Einführung.* Oldenburg (2007).

Studienkurs Algorithmen und Datenstrukturen. Wer Rot-Schwarz-Bäume selbst implementieren will, sollte dieses Buch studieren (Quellen in C).

# C C++ Sprachbeschreibung

## C.1 Lexikalische und syntaktische Grundbegriffe

### C.1.1 Zeichen

Unabhängig von der internen Darstellung besteht ein *Quelltext* aus

**Buchstaben** `A...Z a...z` sowie Unterstrich `_` (gilt als Buchstabe)

**Ziffern** `0 1 2 3 4 5 6 7 8 9`

**Sonderzeichen** `+ - * / % ? : = [ ] ( ) { } , . < > ! | & ~ ^ # ' " \`

**Leerzeichen,** Tabulatoren und Zeilenumbrüche (*white spaces*) dienen zur Gliederung
▶D.2 des Quelltextes.

### C.1.2 Symbole

**Namen** (*Bezeichner*) müssen mit einem Buchstaben (oder Unterstrich[2]) beginnen. Anschließen können sich Buchstaben, Ziffern, Unterstriche in beliebiger Folge. Die Länge von Bezeichnern ist nicht beschränkt.[3] Groß- und Kleinbuchstaben werden unterschieden. Bezeichner vor runden Klammern werden als *Funktionsnamen* interpretiert. Als *Schlüsselwörter* sind reserviert:

```
alignas     continue     friend       register           true
alignof     decltype     goto         reinterpret_cast   try
asm         default      if           return             typedef
auto        delete       inline       short              typeid
bool        do           int          signed             typename
break       double       long         sizeof             union
case        dynamic_cast mutable      static             unsigned
catch       else         namespace    static_assert      using
char        enum         new          static_cast        virtual
char16_t    explicit     noexcept     struct             void
char32_t    export       nullptr      switch             volatile
class       extern       operator     template           wchar_t
const       false        private      this               while
constexpr   float        protected    thread_local
const_cast  for          public       throw
```

An bestimmten Stellen haben die Namen `final` und `override` besondere Bedeutung. Operatornamen sind für einige Operationszeichen reserviert (Tab. 3). Implementationsabhängig können weitere Namen reserviert sein.

[2] Vorsicht! Viele vordefinierte und systemabhängige Bezeichner beginnen / enden mit _

[3] Linker können die Zahl signifikanter Zeichen einschränken.

Tabelle 3: Reservierte Operatornamen.

| | | | |
|---|---|---|---|
| `and` | `&&` | `and_eq` | `&=` |
| `bitand` | `&` | `not_eq` | `!=` |
| `bitor` | `\|` | `or_eq` | `\|=` |
| `compl` | `~` | `xor_eq` | `^=` |
| `not` | `!` | | |
| `or` | `\|\|` | | |
| `xor` | `^` | | |

**Zahlen** beginnen mit einer Ziffer. *Ganzzahlen* sind unterschiedlich darstellbar:

| Darstellung | Basis | Ziffern | Anfangszeichen | Beispiel |
|---|---|---|---|---|
| dezimal | 10 | 0...9 | keine `0` | `127` |
| binär | 2 | 0...1 | `0b` | `0b101010` |
| oktal | 8 | 0...7 | `0` | `0177` |
| hexadezimal | 16 | 0...9, a...f , A...F | `0x` | `0x7F` |

Durch Anfügen von `l` oder `L` erhält die Konstante den Datentyp `long int`. Durch `u` oder `U` am Ende wird die Zahl als vorzeichenlos interpretiert.[4] Sonst wird stets der kleinstmögliche vorzeichenbehaftete ganzzahlige Typ angenommen. Hochkomma kann als Stellentrennzeichen eingesetzt werden: ►C.2.1

```
65 255L  65535U  -1UL  4'294'967'295
```

*Gleitkommazahlen* enthalten einen Dezimalpunkt, einen Exponenten[5] oder beides. Sie haben den Typ `double`. Mit `f` oder `F` am Ende wird der Typ `float` erzwungen, mit `l` oder `L` der Typ `long double`:

```
123.0     0.123     1.602E-19f
123.      .123      299792.458e+8L
```

**Hochkomma und Gänsefüßchen** schließen einzelne Zeichen `'a'` bzw. *Zeichenketten* `"Hallo"` ein. In Zeichenketten darf kein Zeilenende auftreten. Aufeinander folgende durch *white space* getrennte Zeichenketten werden zu einer verschmolzen:

```
"Das ist eine" "Zeichenkette"
```

Durch vorangestelltes `L` erfolgt die Codierung als *wide character*, durch `u8`, `u` oder `U` als *Unicode* (UTF-8, 16 bzw. 32 bit). Einige Zeichen müssen durch Kombination mit dem *Backslash* `\` gebildet werden (Tab. 4). Bei einem *raw string* werden keine Ersetzungen mit dem Backslash durchgeführt, der eigentliche Inhalt wird in eine Begrenzerfolge mit runden Klammern eingeschlossen: `R"delim(...)delim"`.

[4] Die meisten Computer verwenden für negative Werte die Darstellung im Zweierkomplement.
[5] Die Zahl `1.6E-19` ist zu lesen als $1,6 \cdot 10^{-19}$.

Tabelle 4: Sonderzeichen.

| | | |
|---|---|---|
| `\a` | alert | Warnton bei Bildschirmausgabe |
| `\b` | backspace | löscht vorhergehendes Zeichen |
| `\f` | form feed | Blattvorschub (bei Druckern) |
| `\n` | new line | Zeilenumbruch |
| `\r` | return | rückt zum Zeilenanfang (ohne Zeilenumbruch) |
| `\t` | tab | rückt zur nächsten Tabulatorposition |
| `\\` | backslash | druckt `\` |
| `\v` | vertical tab | rückt eine Zeile nach unten |
| `\'` | quote | druckt Hochkomma `'` |
| `\"` | double quote | druckt Gänsefüßchen `"` |
| `\0` | nul | Endekennung von Zeichenketten |
| `\`*ooo* | octal value | Zeichenwert oktal (1...3 Oktalziffern *ooo*) |
| `\x`*hh* | hex value | Zeichenwert hexadezimal (1...2 Hex-Ziffern *hh*) |
| `\u`*hhhh* | wide char | Unicode-Zeichen (`wchar_t char16_t char32_t`) |
| `\U`*hhhhhhhh* | | (4 oder 8 Hex-Ziffern *hhhh*) |

### C.1.3 Weitere syntaktische Einheiten

**Kommentare** `// bis zum Zeilenende oder`
`/* mit beidseitigem Begrenzer */`
sind nicht schachtelbar und werden bei der Programmübersetzung ignoriert. Sie sind nur für die menschliche Kommunikation bestimmt.

**Semikolon** `;`
schließt jede *Anweisung* ab. Was vor dem Semikolon steht, heißt *Ausdruck* (der auch leer sein kann). *Ausdrücke* sind u. a. Deklarationen, Funktionsaufrufe, Zuweisungen.

**Geschweifte Klammern** `{ }`
umschließen *Blöcke* (Deklarations- oder Anweisungsfolgen). Sie können auch leer sein und sind ineinander schachtelbar.

**Doppelkreuze** `#`
►C.7 am Zeilenanfang leiten Anweisungen für den *Präprozessor* ein und gehören (strenggenommen) nicht zur Sprache. Allerdings kommt kaum ein Programm ohne sie aus.

## C.2 Datentypen, Variablen, Operationen

### C.2.1 Elementare Datentypen

Neben `void` als Markierung für leere Plätze[6] gibt es die Grundtypen:

| | |
|---|---|
| `bool` | mit zwei Werten `true` und `false` |
| `char wchar_t` | Zeichen |
| `int` | Ganzzahlen in maschinentypischer Größe[7], |
| `float` | einfach genaue und |
| `double` | doppelt genaue Gleitkommazahlen. |

**Modifikationen** sind für einige Typen erlaubt: `int`'s existieren als
`short int` (mindestens 2 Byte) und
`long int` (mindestens 4 Byte).
Die Angabe `int` kann dann weggelassen werden. `long double` kann noch mehr Stellen als `double` besitzen (IEEE-Extended-Format). Durch die Modifikatoren `signed` und `unsigned` vor `char` und `int` werden Werte mit gesetztem höchstwertigem Bit unterschiedlich interpretiert (heute übliches *Zweierkomplement*). Ohne Angabe gelten `int` als `signed`. Ob `char` als `signed char` oder `unsigned char` gelten, hängt vom Compiler ab und damit auch ihre Ganzzahlrepräsentation:

| | | | |
|---|---|---|---|
| `signed char` | `0..127` | `-128..-1` | (Zweierkomplement) |
| `unsigned char` | `0..127` | `128..255` | |
| hexadezimal | `0..0x7F` | `0x80..0xFF` | (interne Darstellung) |

**Speicherbedarf und Wertebereich** sind hardware- und implementationsabhängig. Zugesichert wird für die Speichergröße in Byte:[8]

$$1 \equiv \texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

Für die *Wertebereiche* der Typen auf einem 32-Bit-System könnte gelten:

| Typ | Byte | Wertebereich | | Genauigkeit |
|---|---|---|---|---|
| `signed char` | 1 | $-128$ | 127 | |
| `unsigned char` | 1 | 0 | 255 | |
| `short` | 2 | $-32768$ | 32767 | |
| `unsigned short` | 2 | 0 | 65535 | |
| `long` | 4 | $-2147483648$ | 2147483647 | |
| `unsigned long` | 4 | 0 | 4294967295 | |
| `float` | 4 | $\pm 1.2 \cdot 10^{-38}$ | $\pm 3.4 \cdot 10^{38}$ | 7 Stellen |
| `double` | 8 | $\pm 2.3 \cdot 10^{-308}$ | $\pm 1.7 \cdot 10^{308}$ | 15 Stellen |
| `long double` | 10 | $\pm 1 \cdot 10^{-4927}$ | $\pm 1 \cdot 10^{4932}$ | 19 Stellen |

[6] Eine *Funktion* `void f()` liefert keinen Rückgabewert.

[7] Maschinentypisch ist etwa die Größe eines Prozessorregisters, `char16_t char32_t` und in `<cstdint>` definierte Typen wie `int8_t int_least64_t uint128_t` kommen zum Einsatz, wenn der exakte Speicherbedarf bzw. Wertebereich von Bedeutung ist.

[8] Ein `char` ist 1 Byte groß, kann aber `CHAR_BIT` Speicher belegen. So hat auf einer DEC-20 der kleinste addressierbare Speicherbereich 36 Bit. Es passen also `char`, `short`, `long` und `float` hinein.

Tabelle 5: Informationen in `std::numeric_limits<T>`.

| Komponente | Bedeutung |
|---|---|
| `is_specialized` | Limit-Angaben für diesen Typ verfügbar |
| `min()` | kleinster Wert, für Gleitkommatypen kleinster positiver Wert |
| `max()` | größter Wert |
| `digits` | Anzahl der Ziffern im Basissystem |
| `digits10` | Anzahl der Ziffern im Dezimalsystem |
| `is_signed` | ist vorzeichenbehaftet |
| `is_integer` | ist ganzzahlig |
| `is_bounded` | ist beschränkt |
| `is_exact` | alle Werte sind exakt darstellbar |
| `is_modulo` | Summe zweier positiver Werte kann negativ sein (Überlauf) |
| `radix` | Zahlenbasis (meist 2) |
| `min_exponent` | kleinste Ganzzahl $e$ mit `radix`$^e$ im Wertebereich |
| `min_exponent10` | kleinste Ganzzahl $e$ mit $10^e$ im Wertebereich |
| `max_exponent` | größte Ganzzahl $e$ mit `radix`$^e$ im Wertebereich |
| `max_exponent10` | größte Ganzzahl $e$ mit $10^e$ im Wertebereich |
| `has_infinity` | Darstellung für $\infty$ verfügbar |
| `infinity()` | $\infty$ |
| `epsilon()` | kleinste Zahl $\varepsilon$ mit $1 + \varepsilon > 1$ (bei Gleitkommazahlen) |
| `round_error()` | maximaler Rundungsfehler |
| `round_style` | Art der Rundung: `round_indeterminate`, `round_toward_zero`, `round_to_nearest`, `round_toward_infinity` oder `round_toward_neg_infinity` |

Die Template-Klasse `numeric_limits<T>` (Tab. 5) bietet eine einheitliche Schnittstelle für Größeninformationen, die auch für eigene numerische Typen `T` erweitert werden kann:

```
#include <iostream>
#include <limits>

int main()
{
  std::cout << std::numeric_limits<int>::digits << " Bits, "
            << std::numeric_limits<int>::min()  << "..."
            << std::numeric_limits<int>::max()  << '\n';
}
```

liefert Informationen zum Datentyp `int`, z. B. auf einem 32-Bit-System

```
31 Bits, -2147483648...2147483647
```

### C.2.2 Variablen

**Die Deklaration** ⟨*typ*⟩ ⟨*name*⟩;
legt für jeden Variablennamen fest, welchen Datentyp er besitzt, bevor er benutzt werden kann. Durch Einrichtung des Speichers, der den Wert der Variable aufnimmt, wird die Variable *definiert*. Dabei kann die Variablen *initialisiert* werden.[9] Der Typ einer `auto`-Variable wird durch ihren Initialisierer festgelegt. Mit `decltype` kann man sich auf den Typ von Variablen und Ausdrücken beziehen.

```
int ganzzahl;
auto c = 'A';    // char
float f = 1.23f;
long i = 0, j;
decltype(i) k = i; // long
```

**Der Gültigkeitsbereich** von Variablen wird durch den Kontext festgelegt, in dem die Deklaration erfolgt. Variablen und Konstanten sind erst nach (unterhalb) ihrer Deklaration im Quelltext bekannt und verwendbar. Außerhalb von Blöcken definierte Variablen sind *global* gültig. Globale Variablen gleichen Namens in verschiedenen Dateien sind voneinander unabhängig (*file scope*[10]), sofern auf sie nicht `extern` Bezug genommen wird.

In einem Block `{...}` festgelegte Variablen sind *lokal* zu diesem Block, außerhalb des Blocks ist der Variablenname unbekannt (*block scope*). Gleichnamige lokale Variablen in verschiedenen Blöcken sind unabhängig voneinander. Innere lokale Variablen verdecken äußere globale und lokale Variablen gleichen Namens. *Funktionsparameter* gelten als lokal deklarierte Variablen. ►C.6

```
#include <iostream>

float x = 2.5;         // globales x

float sqr(float x) { return x*x; } // Parameter x lokal

int main()
{
  float x2 = sqr(x);   // globales x
  {
    float x = x2;      // lokales x
    std::cout << x << ' ' << ::x <<  '\n';
  }          // ^ lokal     ^^^ global

  return x2 == x*x;   // wieder globales x
}
```

[9] Ohne Startwert erhalten statische Variablen den Wert 0, nichtstatische sind undefiniert (Müll).
[10] Programme können aus mehreren, getrennt übersetzbaren Quelldateien bestehen.

**Die Lebensdauer** und damit der Speicherplatzbedarf globaler Variablen erstreckt sich über die gesamte Programmdauer (*statische* Variablen). Für lokale Variablen wird beim Eintritt in den Block *automatisch* Speicherplatz auf dem Stapel reserviert und beim Verlassen des Blocks wieder freigegeben. Damit geht auch der gespeicherte Wert verloren.

**Eine Speicherklasse** `extern static`
kann einer Deklaration vorangestellt werden.

`extern` erklärt, dass der deklarierte Name an anderer Stelle global definiert ist, z. B. in einer anderen Quelldatei des Programms.[11]

`static` vor einer globalen Deklaration versteckt diese vor dem `extern`-Zugriff aus anderen Quelldateien.[12] Als `static` deklarierte lokale Variablen werden beim Programmstart initialisiert und behalten ihren Wert auch nach dem Verlassen einer Funktion. Eine globale Deklaration ist in solchen Situationen unnötig:

```
int noch_ein_Bier_bitte()
{
  static int Biere = 0;
  return ++Biere;     // Anzahl der bestellten Biere
}
```

**Referenzvariablen** ⟨*typ*⟩ `&` ⟨*aliasname*⟩ `=` ⟨*variable*⟩`;`
definieren feste Aliasnamen, die für ihre Lebensdauer fest mit der Speicheradresse einer Variable verbunden sind. Sie erlauben die Übergabe von *Referenzparametern* an Funktionen (Zurückliefern von Rechenergebnissen aus Funktionen). Seltener werden Referenzen solo verwendet (Vermeidung wiederholter aufwendiger Adressberechnungen, Lesbarkeit).

```
void spur(int& diagonalsumme)      // Referenzparameter
{
  int matrix[3][3];                // ein 3x3-Feld
  // ...
  int& linksoben   = matrix[0][0];
  int& mitte       = matrix[1][1]; // oft benutzte Stellen
  int& rechtsunten = matrix[2][2];
  mitte = 15;                      // dasselbe wie matrix[1][1]=15;
  // ...
  diagonalsumme = linksoben + mitte + rechtsunten;
}
```

[11] Globale Variablen besitzen *externe Bindung*. Namen, auf die mit `extern`-Deklarationen Bezug genommen wird, dürfen nur in einer Datei definiert werden. Ohne `extern` wären gleichnamige globale Variablen in verschiedenen Quelltexten unabhängig voneinander.

[12] Diese Verwendung von `static` wird im Standard missbilligt. Stattdessen sollte durch unbenannte Namensbereiche `namespace` {...} der externe Zugriff verhindert werden. Generell sind globale Variablen problematisch.

### C.2.3 Konstanten und flatterhafte Variablen

Durch Voranstellen von `const` lässt sich (unabsichtliches) Ändern des Variablenwertes nach der Initialisierung verbieten:[13]

```
const double euro = 1.95833; // in DM
```

Konstante Referenzen oder *Zeiger* müssen nicht unbedingt auf Konstanten zeigen. Sie sind vielmehr eine „freiwillige Selbstverpflichtung“, die Variable nicht zu ändern:

```
int jahr = 1999;
const int& cref = jahr; // nur lesen, nicht schreiben
```

Als `mutable` deklarierte Variablen dürfen auch dann geändert werden, wenn die Instanz der umgebenden Struktur oder Klasse logisch konstant ist:

```
struct Count { mutable int counter; };

void f(const Count& s) { ++s.counter; }
```

Die Deklaration einer Variable als `volatile` (flüchtig) verhindert Optimierungsversuche des Compilers. Bei jeder Auswertung der Variable muss der Speicher neu ausgelesen werden. Ihr Inhalt kann sich durch Einflüsse außerhalb des Programm ändern.

```
volatile int users = 0;
```

**Aufzählungen** `enum` ⟨*Typname*⟩$_{opt}$ { ⟨*Liste von Konstanten*⟩ } ⟨*Variablenliste*⟩$_{opt}$ ; sind Sammlungen ganzzahliger Konstanten. denen ein gemeinsamer Typname zugewiesen werden kann. Die Konstanten werden mit 0 beginnend aufsteigend numeriert, falls keine anderen Werte angegeben werden. Der gleiche Wert kann auch mehrmals vorkommen.

```
enum Muenzen { einer=1, zweier, fuenfer=5,
               groschen=10, fuffziger=50 } Kleingeld;
```

Werte vom Aufzählungstyp lassen sich problemlos einer Ganzzahlvariablen zuweisen, umgekehrt jedoch nicht (explizite Typumwandlung erforderlich). Damit können auch die Operatoren `++` und `--` nicht angewendet werden. Typsichere Aufzählungen (*strong enums*) lassen auch die implizite Umwandlung in einen Ganzzahltyp nicht zu:

```
enum class TriState : char { unknown, no, yes };
TriState s = TriState::unknown;
// char c = s;    // nicht erlaubt
char c = char(s); // ok
```

Ein Grundwertebereich kann angegeben werden.

[13] Der direkte Änderungsversuch erzeugt einen Übersetzungsfehler. C-Programmierer sagen trotzdem zur `const`-Variablen ungern Konstante, weil sie
a) wissen, wie man `const` umgehen kann,
b) der Drang, das zu tun, beliebig groß werden kann und
c) die Bezeichnung Konstante aus historischen Gründen vom Präprozessor belegt ist.

**`constexpr`-Ausdrücke** sind konstant, aber können im Gegensatz zu `const`-Variablen wie *Literale* (z.B. Zahlkonstanten) schon zur Übersetzungszeit ausgewertet werden. Zur ihrer Berechnung werden ebenfalls nur solche Werte herangezogen, die schon bei der Übersetzung festgelegt sind:

```
constexpr int size = sizeof(int);
constexpr const char* s = "Hallo"; // const char* const s
```

►C.9

Auch Rückgabewerte als `constexpr` markierter Funktionen und Konstruktoren sind vom Compiler auswertbar, sofern zu ihrer Berechnung nur Anweisungen herangezogen werden, die vom Compiler bei der Übersetzung auswertbar sind[14]:

```
constexpr int sqr(int x) { return x*x; }
constexpr int vier = sqr(2);
char arr[sqr(4)];

struct Punkt
{
  int x, y;
  constexpr Punkt(int a = 0, int b = 0) : x(a), y(b) {}
};

constexpr Punkt p = { 2, 3 };
```

**Zusicherungen** mit `static_assert` liefern schon beim Übersetzen einen Fehler, wenn der logische Ausdruck nicht zutrifft. Dabei können als `constexpr` markierte Ausdrücke in die Berechnung einfließen:

```
static_assert(sizeof(int) == 4, "keine 32bit-Architektur");
static_assert(vier == sqr(2)); // ab C++17 auch ohne Fehlerbeschreibung
```

Zur Laufzeit kann die Einhaltung von Bedingungen mit

```
assert(auswahl < 4); // #include <cassert>
```

geprüft werden. Trifft der Ausdruck nicht zu, wird der Programmlauf wird mit einer (implementierungsabhängigen) Fehlermeldung beendet:

```
Assertion "auswahl < 4" failed in file: assert.cpp, Line 29.
```

[14] Die Regeln hierfür wurden seit der Einführung in C++11 mehrfach gelockert.

### C.2.4 Operationen

**Zuweisungen** ⟨*Lvalue*⟩ `=` ⟨*Ausdruck*⟩`;`
ändern Variablenwerte. Die linke Seite (das Ziel der Zuweisung) muss nicht unbedingt ein Variablenname sein, vielmehr ein Ausdruck[15], der einen modifizierbaren Speicherplatz besitzt:

```
int j=1;
char p[2][4] = {"du", "ich"};
*p[j] = 'a';   // Ergebnis: "ach"
```

Eine Zuweisung ist ebenfalls wieder ein Ausdruck. Ihr Wert ist gleich dem des Linkswertes nach der Zuweisung. Zuweisungen können dadurch in einer Kette erfolgen oder an Stellen auftreten, wo man sie nicht erwartet. Mehrere Zuweisungen werden von rechts nach links abgearbeitet (*rechtsassoziativ*).

```
a = b = c = 123;
x = sqrt(a=3*x); // kryptisch!
```

**Verbundzuweisungen** ⟨*Lvalue*⟩ ⟨*Binäroperator*⟩`=` ⟨*Ausdruck*⟩`;`
mit den Verbundoperatoren (Operatornamen darunter)

```
+= -= *= /= %= &=     |=    ^=     <<= >>=
               and_eq or_eq xor_eq
```

sind *Kurzschreibweisen* für binäre (zweistellige) arithmetische oder bitweise Operationen mit anschließender Zuweisung an den linken Operanden:

```
j += 10;       // j = j+10;      => 10 hinzu addieren
j *= 1+2;      // j = j*(1+2);   => verdreifachen, beachte Rangfolge!
*p[j] += 3;    // *p[j] = *p[j]+3;
```

Die Speicheradresse des linksseitigen Ausdrucks der Operationen muss auf diese Weise nur einmal ermittelt werden.

**Arithmetische Operationen** `x+y  x-y  x*y  x/y  x%y`
zur Addition, Subtraktion und Multiplikation sind auf alle Grunddatentypen anwendbar. Die Division liefert einen Gleitkommawert, falls mindestens ein Operand vom Gleitkommatyp ist. Bei ganzzahligen Operanden ist das Ergebnis der ganzzahlige Anteil des Bruches. Der Divisionsrest lässt sich bei ganzzahligen Operanden mit dem Modulo-Operator `%` berechnen:

```
std::cout << 22/3.0 <<' '<< 22.0/3 <<' '<< 22/3 <<' '<< 22%3 <<'\n';
// Ausgabe:  7.3333...      7.3333...      7            1
```

[15] Ein Linkswert ist ein Ausdruck, der links von `=` stehen darf (engl. *Lvalue*).

*Typumwandlungen* erfolgen *implizit* vor Operationen bei Operanden unterschiedlichen Typs, immer zum „größeren“ Typ hin:

```
double f(char ch, int i, float f, double d)
{
   return  i%ch +  f/i  - d*i;
        // int  + float - double
        //    float     - double
        // also:      double
}
```

*Explizite* Typumwandlung ⟨*typ*⟩(⟨*ausdruck*⟩) oder (⟨*typ*⟩)⟨*ausdruck*⟩[16]
kann manchmal notwendig sein, um etwa eine bestimmte Art der Division zu erzwingen:

```
double dezimal(int x, int y)  // Beispiel: x = 3, y = 2
{
  return double(x)/y;         // 3.0/2 = 1.5 statt 1
}
```

Typumwandlungen bei der Zuweisung und Wertrückgabe in Funktionen erfolgen automatisch und ohne Warnung, falls rechte und linke Seite bzw. Audruck und Rückgabetyp typverschieden sind. Der rechte Ausdruck wird in den Typ der linken Seite ohne Warnung umgewandelt:

```
void g(int i, long l, float f, double d)
{
  signed char c = -128; // signed char  <== int
  d = f = l = i = c;    // double <== float <== long <== int <== char
}                       // alle -128
```

Passt der Wert des Ausdrucks in den Wertebereich des Lvalues, bleibt der Wert erhalten, auch Vorzeichen werden korrekt behandelt. Bei Umwandlung von Gleitkommazahlen in Ganzzahlen wird der gebrochene Anteil abgeschnitten:

```
int pi = 3.1415926535; // pi=3!
```

Passt der rechtsseitige Ausdruck nicht in den Wertebereich der linken Seite, so werden bei ganzzahligen Typen höherwertige Bits weggelassen, bei Gleitkommatypen ist das Ergebnis nicht definiert:[17]

```
signed char c = 65535; // c= -1
```

Umwandlungen zwischen `signed` und `unsigned` erfolgen über das Zweierkomplement.

[16] C-Notation, siehe auch bei Typecast-Operationen
[17] Das ist eine verharmlosende Umschreibung für maschinenabhängig.

**Inkrement- und Dekrementoperatoren** `++` `--` erhöhen bzw. senken den Wert ihres Lvalue-Operanden um 1:

```
x++; --y;   // bedeuten:  x = x+1; y = y-1;
```

In zusammengesetzten Ausdrücken hängt von der Stellung des Operators ab, ob der alte oder der neue Wert des Operanden weiterverwendet wird. *Präfix-* (vorangehende) Operatoren verändern ihren Operanden *vor* der Weitergabe des Wertes. *Postfix-* (nachgestellte) Operatoren liefern den alten Wert zur weiteren Auswertung und verändern *nachher* den Wert ihres Operanden:

```
x = 10;
y = x++;  // y = x;   x = x+1; also: x = 11 y = 10
y = ++x;  // x = x+1; y = x;   also: x = 12 y = 12
```

**Vergleiche und logische Verknüpfungen** liefern als Ergebnis `false` oder `true`[18]:

| | | | | | |
|---|---|---|---|---|---|
| `x < y` | x kleiner als y | | `a && b` | a und b | `a and b` |
| `x <= y` | x kleiner/gleich y | | `a \|\| b` | a oder b | `a or b` |
| `x == y` | x gleich y | | `!a` | nicht a, (`a == 0`) | `not a` |
| `x != y` | x nicht gleich y | `x not_eq y` | | | |
| `x >= y` | x größer / gleich y | | | | |
| `x > y` | x größer als y | | | | |

Logische Ausdrücke werden, von links beginnend, nur soweit ausgewertet, bis das Ergebnis feststeht: Der Ausdruck `x != 0 && 1/x < 1` vermeidet eine Division durch Null bei `1/x`.

**Bitweise Operationen** werden auf allen Bits ganzzahliger Typen ausgeführt:

| | | |
|---|---|---|
| `i&j` | `i bitand j` | bitweise UND |
| `i\|j` | `i bitor j` | bitweise ODER |
| `i^j` | `i xor j` | bitweise XOR (Exklusiv-Oder) |
| `~j` | `compl j` | Bitkomplement (vertauscht 0 und 1) |
| `i << n` | | Linksschieben um `n` Bits[19] |
| `i >> n` | | Rechtsschieben um `n` Bits |

**Der bedingte Ausdruck** ⟨*Ausdruck1*⟩ `?` ⟨*Ausdruck2*⟩ `:` ⟨*Ausdruck3*⟩
mit dem dreistelligen Entscheidungsoperator kann Verzweigungen kompakt darstellen. Ist ⟨*Ausdruck1*⟩ wahr (ungleich Null), so wird der Wert von ⟨*Ausdruck2*⟩ berechnet, sonst ⟨*Ausdruck3*⟩. ⟨*Ausdruck2*⟩ und ⟨*Ausdruck3*⟩ müssen typgleich sein:

```
min = x<y ? x : y;   // Minimum von x und y
x<y ? x : y = 100;   // Variable mit kleinerem Wert wird 100 zugewiesen
```

[18] Jeder Wert ungleich 0 gilt als wahr: C++-Programmierern fällt „Ja“-Sagen leichter.

[19] Beim Bitschieben muss der rechte Operand positiv sein. Das Ergebnis des Rechtsschiebens bei negativen `signed`-Werten des linken Operanden ist maschinenabhängig.

**Der Kommaoperator** ⟨*Ausdruck1*⟩`,` ⟨*Ausdruck2*⟩ ...
verbindet mehrere Ausdrücke zu einer Liste und wertet diese von links nach rechts aus. Der letzte Wert ist der Wert des Ausdrucks, alle vorhergehenden werden verworfen. Dies erlaubt mehrere „Anweisungen“ dort, wo nur ein Ausdruck stehen darf, z. B. in einem Schleifenkopf:

```
for(i = 0, j = 10, n = 0; i < j; ++i, --j) ++n;
```

**Typinformationen** zum *Speicherbedarf* von Datentypen und Variablen lassen durch `sizeof(`⟨*Typ*⟩`)` bzw. `sizeof` ⟨*Ausdruck*⟩ ermitteln. Die Abfrage `typeid(`⟨*Ausdruck*⟩`)` erzeugt ein `std::type_info`-Objekt, welches zur Analyse verwendet werden kann:

```
#include <iostream>
#include <typeinfo>

void typnamen()
{
  std::cout << typeid(int).name() << ' ' << sizeof(int) << '\n';
}
```

**Typecasting** `(`⟨*Typ*⟩`)`⟨*Ausdruck*⟩
erzwingt die Umwandlung des ⟨*Ausdruck*⟩ in einen anderen ⟨*Typ*⟩. Die Sinnhaftigkeit des Dateninhaltes wird dabei aber nicht garantiert. Typecasts setzen das Typen- und Schutzsystem von C++ außer Kraft und sind im Quelltext schwer zu erkennen. Um genauer anzugeben, warum ein *Typecast* notwendig war, sollten die auffälligeren Cast-Formen benutzt werden:

- `static_cast<`⟨*Typ*⟩`>(`⟨*Ausdruck*⟩`)`
  konvertiert verwandte Typen wie verschiedene Zeigertypen, Aufzählungen, ganzzahlige und Gleitkommatypen und erlaubt noch eine minimale Typprüfung,
- `dynamic_cast<`⟨*Typ*⟩`>(`⟨*Zeiger oder Referenz*⟩`)`
  um in einer Klassenhierarchie einen *Downcast* oder *Crosscast* zu erlangen; der Compiler prüft die Zulässigkeit zur Laufzeit, bei Scheitern wird ein Nullzeiger geliefert oder eine `bad_cast`-Ausnahme geworfen,
- `const_cast<`⟨*Typ*⟩`>(`⟨*konstanter Ausdruck*⟩`)`
  um Schreibrechte auf einen konstanten Ausdruck zu erlangen (Einbrecher!),
- `reinterpret_cast<`⟨*Typ*⟩`>(`⟨*Ausdruck*⟩`)`
  konvertiert nicht verwandte Typen wie Zeiger und `int` (ohne Typprüfung):

  ```
  IO_device* d1 = reinterpret_cast<IO_device*>(0xFF00);
  ```

**Weitere Operatoren** (Bereichsauflösung `::` in Klassen und Namensbereichen, Werfen einer Ausnahme `throw`, Freispeicherverwaltung `new delete`, Zeigerzugriff `& *`, Strukturkomponentenzugriff `. -> .* ->*` und Feldelementzugriff `[]`) werden an geeigneter Stelle beschrieben.

Tabelle 6: Rangfolge der Operatoren.

| | | |
|---|---|---|
| Bereichsauflösung | `::` | (höchster Rang) |
| unär (Postfix) | `++ --` ⟨*Typ*⟩`() [] . ->` | |
| unär (Präfix) | `++ -- ! ~ - + * & delete new` | |
| Elementselektion | `.* ->*` | |
| multiplikativ | `* / %` | |
| additiv | `+ -` | |
| Bitschieben | `<< >>` | |
| 3-Wege-Vergleich | `<=>` (ab C++2a) | |
| Vergleich | `< <= > >=` | |
| | `== !=` | |
| bitweise | `&` | |
| | `^` | |
| | `\|` | |
| logisch | `&&` | |
| | `\|\|` | |
| Entscheidung | `? :` | |
| Zuweisung | `= += -= *= /= %= <<= >>= &= \|= ^=` | |
| Liste | `,` | (niedrigster Rang) |

*Effektives Schreiben (oder Lesen) in C ohne Kenntnis dieser Regeln ist unmöglich. Bitte studiere die Rangtabelle jeden Abend beim Zähneputzen.*
– Numerical Recipes

### C.2.5 Auswerteregeln

**Die Rangfolge** (*Präzedenz*) der Operatoren (Tab. 6) lässt sich durch Klammersetzung ändern: `2*(3+5)` statt `2*3+5`. Aufeinander folgende unäre Präfix-Operatoren und Zuweisungen gleichen Ranges binden von rechts nach links (*rechtsassoziativ*), alle anderen Operatoren binden von links nach rechts (*linksassoziativ*):

```
int i = 2+3+5;       // i= (2+3)+5;
x = y += z=5;        // x= (y+=(z=5));
```

**Die Auswertungsreihenfolge** der Operanden ist außer bei `&& || ?: ,` compilerabhängig. Nicht portable und unklare Konstruktionen sind zu meiden: ▶D.1.1

```
int x = 0, i = 1;
v = (x=4) - (--x);  // v = 1 oder v = 5 ?
a[i] = i++;         // a[1] = 1 oder a[2] = 1 ?
```

## C.3 Gruppierung von Daten

### C.3.1 Felder

Eine Anzahl typgleicher Elemente ⟨*Typ*⟩ ⟨*Feldname*⟩`[`⟨*Anzahl*⟩`];`
belegt einen zusammenhängenden Speicher von `sizeof(`⟨*Typ*⟩`)*`⟨*Anzahl*⟩ Bytes:

```
int a[10]; // 10 int-Elemente a[0]..a[9]
```

Der Zugriff auf die einzelnen Elemente `a[0]` bis `a[9]` erfolgt ohne Bereichsprüfung.[20]
Schreibzugriffe außerhalb der Feldgrenzen haben schwer findbare, häufig fatale Folgen:[21]

```
a[-10]  = 5;  // zehnter int vor dem Feldanfang
a[1000] = 16; // tausendster danach (irgendwo)
```

►C.12.2 **Zeichenketten** sind `char`-Felder, die mit einem Nullbyte `'\0'` abschließen:[22]

```
void C_Zeichenketten()
{ char s1[6] = {'H','a','l','l','o','\0'};  // 6 Byte
  char s2[]  = "kurzer Prozess";            // 15 Byte inklusive '\0'
  s2[4] = 0;                                // abschneiden: "kurz"
  cout << "nur noch " << std::string(s2).size() << " Zeichen.\n"; // 4
}
```

**Mehrdimensionale Felder** werden durch Aufreihungen von eckigen Klammern erklärt (Felder von Feldern). Zwischen benachbarten Speicherplätzen ändert sich der rechteste Index am schnellsten („zeilenweise“ Anordnung). Bei der Definition kann das Feld mit Anfangswerten belegt werden:

```
int rechteck[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

Die inneren geschweiften Klammern können bei der Initialisierung weggelassen werden, der äußerste Index ebenfalls — er wird dann automatisch vom Compiler bestimmt. Anderenfalls ist auch unvollständige Initialisierung möglich. Die Belegung der Feldwerte erfolgt dann mit steigendem Index, falls durch innere geschweifte Klammern nichts anderes erzwungen wird:

```
int y[2][3] = {1, 2, 3};   // nur erste Zeile  y[0][j] festgelegt
int z[2][3] = {{1}, {4}};  // nur erste Spalte z[i][0] festgelegt
```

[20] Sogenannte *C-Index-Konvention*. Jedem seine Extrawurst (erschwert Portierungen):

| | | | |
|---|---|---|---|
| BASIC | `a(0)`...`a(`*anz*`)` | zeilenweise | `a(i,j)` |
| C | `a[0]`...`a[`*anz*`-1]` | zeilenweise | `a[i][j]` |
| FORTRAN | `a(1)`...`a(`*anz*`)` | spaltenweise | `a(i,j)` |
| PASCAL | `a[1]`...`a[`*anz*`]` | zeilenweise | `a[i,j]` |

[21] „Wilde“ Veränderung von Daten bis zum Systemabsturz.

[22] Die Klasse `std::string` bietet mehr Sicherheit und Komfort.

### C.3.2 Strukturen

Inhaltlich zusammengehörende Daten unterschiedlicher Typen lassen sich als *Komponenten* unter einem *Strukturnamen* zusammenfassen:

`struct` ⟨*Strukturname*⟩$_{opt}$ `{` ⟨*Komponenten*⟩ `}` ⟨*Variablenliste*⟩$_{opt}$`;`

Vorwärtsdeklarationen

`struct` ⟨*Strukturname*⟩;

können benutzt werden, solange der Compiler keine Information über den Strukturinhalt benötigt, z. B. um einen Zeiger zu bilden.

Die Initialisierung von Strukturvariablen kann bei ihrer Definition erfolgen. Dazu sind die Initialisierungswerte in der Reihenfolge der Komponentendeklaration anzugeben.

Zuweisungen kopieren Strukturen als Ganzes (Byte für Byte), auch enthaltene Felder. Der Zugriff auf einzelne Komponenten erfolgt durch ⟨*Variable*⟩`.`⟨*Komponente*⟩.

```
struct Atom {
  char symbol[3];
  float rel_masse;        // bezogen auf 1u = 1/12 m(12 C 6)
};

const Atom element[] =
{
  "n" , 1.0086,
  "H" , 1.0079,
  "He", 4.0026            // usw.
};

Atom H  = element[1],     // Zuweisung kopiert gesamte Struktur
     He = element[2];

const float u = 1.6605655e-27, // atomare Masseeinheit (in kg)
            c = 2.99792458e+8; // Lichtgeschwindigkeit (in m/s)

float massendefekt   = (4*H.rel_masse-He.rel_masse)*u, // in kg
      fusionsenergie = massendefekt*c*c;               // in Ws
```

Neben Datenfeldern dürfen Strukturen auch *Methoden* zur Manipulation der enthaltenen Daten enthalten (siehe Klassenkonzept).

### C.3.3 Erweiterte Typen

`typedef` ⟨*Typ*⟩ ⟨*neuer Typname*⟩`;`

gibt schon definierten oder gerade deklarierten namenlosen Typen neue Namen:

```
typedef atom Element;
typedef struct { int x,y; char color; } Pixel;
```

### C.3.4 Bitfelder und Vereinigungen

Knapper Speicherplatz oder die Ansteuerung von Geräteschnittstellen erfordern, kleine Informationseinheiten in ein Maschinenwort zu packen. Hinter jeder Komponente wird die Anzahl der zu reservierenden Bits angegeben. Die *Bitfeldkomponenten* verhalten sich wie kleine ganze Zahlen mit oder ohne Vorzeichen. Für Zwischenräume können unbenannte Bitfelder `unsigned :`⟨*Breite*⟩`;` eingeschoben werden. Fast alles an Bitfeldern ist implementierungsabhängig. Bitfeldkomponenten haben keine Speicheradresse und können keine Felder bilden.

```
struct farbe
{
  unsigned vg:4;                // Vordergrundfarbe 0..15
  unsigned hg:3;                // Hintergrundfarbe 0..7
  unsigned bl:1;                // Blinkbit         0..1
};
```

Bei *Vereinigungen* wird der Speicherbereich mehrerer Komponenten überlagert. Je nach Datentyp der Komponenten werden die gleichen Bits unterschiedlich interpretiert. Die Größe der Vereinigung (`union`) entspricht dem größten enthaltenen Typ.[23] Typen mit nichttrivialen Konstruktoren bzw. Destruktoren gehören nicht in eine `union`: ab C++17 gibt es dafür `std::variant<Typelist...>`. Die Nutzung einer `union` zur Typkonversion ist gefährlich, weil nicht portabel:

▶C.19.3

```
union
{
  unsigned char color;
  struct farbe bits;
} attrib;                       // Farbattribut des Textbildschirms

int main()
{
  attrib.bits.vg = 14;          // YELLOW
  attrib.bits.hg = 1;           // BLUE
  attrib.bits.bl = 1;           // attrib.c=YELLOW+16*BLUE+128

  char far* const screen = (char far*) 0xB8000000; // IBM-PC unter DOS

  for (int i = 0; i < 256; ++i)
  {
    screen[2*i]   = i;          // kompletter IBM-Zeichensatz
    screen[2*i+1] = attrib.color; // gelb auf blau blinkend
  }
}
```

[23] Dasselbe sind variante Records (PASCAL) oder EQUIVALENCE-Bereiche (FORTRAN).

## C.4 Zeiger

Jede Speicherzelle des Rechners, in der Daten (Konstanten und Variablen) abgelegt werden, hat eine bestimmte Nummer (*Adresse*). Diese Adresse kann in einer Variable abgelegt werden und als *Zeiger* dienen, um *indirekt* auf Daten zuzugreifen, etwa um

- durch Datenfelder zu laufen (geht mit Zeigern schneller als über Indizes),
- den Platzbedarf für Daten im Freispeicher dynamisch anzupassen, ►C.11
- den Rechnerspeicher direkt zu manipulieren (Vorsicht!).

### C.4.1 Deklaration und Initialisierung

Die Deklaration ⟨*Typ*⟩ `*`⟨*Zeiger*⟩`;` ist auf zweierlei Weise lesbar:

1. ⟨*Zeiger*⟩ enthält die Adresse einer ⟨*Typ*⟩-Variable (ist ein Zeiger auf diese).
2. Der Ausdruck `*`⟨*Zeiger*⟩ besitzt den ⟨*Typ*⟩ (sozusagen als „Muster“).

Bei der Benutzung müssen Zeiger gültige Adressen enthalten (z. B. über den Adressoperator `&`). Die Initialisierung kann auch schon bei der Definition erfolgen. Zeiger ohne gültige Adresse sollten den speziellen Wert `NULL` erhalten:

```
int i;
int *ip;      // Ausdruck *ip ist vom Typ int
ip = &i;      // Zeiger ip bekommt Adresse, zeigt auf i

// oder:
int *ip = &i; // sofortige Initialisierung des Zeigers ip
*ip = 7;      // Ergebnis: i=7

ip = NULL;    // ip zeigt jetzt nirgendwohin
```

### C.4.2 Fehlerquellen (wild pointer)

*Weil Zeiger solche Alpträume verursachen, ist es besser, niemals einen zu erzeugen.*
– Herbert Schildt: C — The Complete Reference, Addison-Wesley

Zeiger bergen die Gefahr fataler Programmierfehler in sich, die schwer zu finden sind. Sie können Programmabstürze verursachen, unkontrolliert Daten verfälschen, oder zunächst nicht einmal bemerkt werden.

1. Nicht initialisierte, „wilde“ Zeiger enthalten Müll! Das Schreiben auf unkontrollierte Adressen gleicht russischem Roulette:

   ```
   int x = 10;
   int *p;
   *p = x; // Schreiben an nicht festgelegte Adresse
   ```

2. Bei einer möglichen Verwechslung eines Zeigers mit einer Ganzzahl erzeugt der Compiler eine Fehlermeldung, deren Auflösung tiefes Nachdenken erfordert:

```
int x = 10;
int *p;
p = x; // Typ unvereinbar: p = &x oder *p = x?
```

3. Adressen lokaler Variablen sind nach Verlassen der Funktion ungültig. Das ergibt „baumelnde Zeiger“:

```
int *p;
void f()
{
  int x = 10;
  p = &x;
}  // Variable x wird hier freigegeben, Adresse ungueltig
```

### C.4.3 Zeigerarithmetik

**Zeiger und Feldnamen** sind eng verwandt. Feldnamen können als die Adresse des Anfangselementes aufgefaßt werden.[24] Als konstante Zeiger auf das Anfangselement können sie nicht versetzt werden. Die Anweisung `a = &p[1];` wäre ein Fehler. Zeiger können benutzt werden, um indirekt auf die Feldelemente zuzugreifen:

```
int a[10];
int *p = &a[0];  // p = a;
p[1] = 10;       // Zuweisung an a[1], weil p == a
```

**Addition und Subtraktion ganzzahliger Werte** zu Zeigern liefern wieder Zeiger (vom selben Typ). `p+i` zeigt auf das `i`-te Element nach `p`. Der Ausdruck `p[i]` ist äquivalent zu `*(p+i)`. Inkrement- und Dekrementoperatoren verschieben Zeiger ein Feldelement vor bzw. zurück. Alle Zeigeroperationen berücksichtigen automatisch die Größe der Objekte, auf die gezeigt wird. Für `void*` ist deshalb keine Zeigerarithmetik durchführbar.

```
p += i;  // p geht i Elemente weiter
p++;     // Zeiger zum folgenden Element
--p;     // Zeiger auf das Element davor
```

**Differenzen und Vergleiche zweier Zeiger** sind nur sinnvoll, wenn beide Zeiger auf Elemente desselben Speicherbereiches (Feldes) zeigen. Vergleiche zweier Zeiger liefern Wahrheitswerte. So ist `p < q` wahr, wenn `p` auf ein früheres Element als `q` zeigt (`p+i = q` mit `i > 0` gilt). Die Differenz zweier Zeiger liefert einen ganzzahligen Wert `i = q-p` entsprechend `q = p+i`:

[24] Solange die Definition sichtbar ist. enthalten sie auch noch die Größeninformation (*pointer decay*).

```
int strlen(const char p[])
{
  const char *q = p;
  while (*q) q++;
  return q - p;
}
```

**Felder mit frei wählbaren Grenzen** sind damit trotz C-Index-Konvention möglich. Eine Prüfung der Feldgrenzen erfolgt dennoch nicht.

```
const int lo = -12;
const int hi =  10;
int zerobase[hi-lo+1];
int *const a = zerobase-lo; // a ist konstanter Zeiger auf int

a[-12] = 3; // Feldelemente a[lo]..a[hi] verwendbar
a[10]  = 5;
```

### C.4.4 Besondere Zeiger

**Zeiger auf Strukturen** erlauben den Zugriff auf Komponenten über (*⟨*Zeiger*⟩*).⟨*Komponente*⟩ oder ⟨*Zeiger*⟩->⟨*Komponente*⟩.

**Zeigerfelder** statt mehrdimensionaler Felder können Speicherplatz einsparen:[25]

```
char t1[][11]= {"Sonntag", "Montag", "Dienstag", "Mittwoch",
                "Donnerstag", "Freitag", "Sonnabend"};
char *t2[]   = {"Sonntag", "Montag", "Dienstag", "Mittwoch",
                "Donnerstag", "Freitag", "Sonnabend"};
```

**Zeiger auf Funktionen** erhalten den Namen (die Adresse) einer definierten Funktion zugewiesen. Ergebnis- und Argumenttypen sollten übereinstimmen: ►C.6

```
#include <cmath>      // definiert sin(), cos(), tan() etc.

typedef double (*funktion)(double);  // Zeiger auf eine Funktion
funktion f[] = { std::sin, std::cos, std::tan };
                                     // Feld von Funktionszeigern

double auswahl_funktion(int num, double x)
{
  if (num < 0 || num > 2) return 0;
  return f[num](x);                  // sin(x), cos(x) oder tan(x)
}
```

[25] Das zweidimensionale Feld `t1` reserviert 77 Zeichen, `t2` nur ein Feld für 7 Zeiger, dennoch ist `t1[6][5]` und `t2[6][5]` jeweils das `b` in `Sonnabend`.

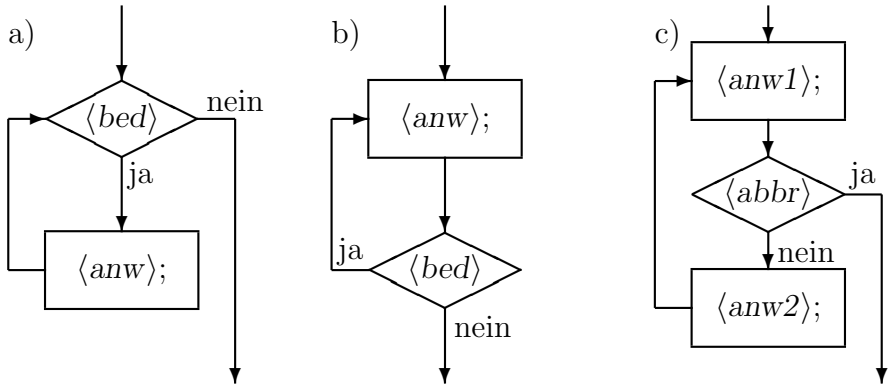



Abbildung 1: Schleifen mit Test (a) am Anfang, (b) am Ende und (c) in der Mitte.

## C.5 Steueranweisungen

### C.5.1 Schleifen

Bestimmte Programmteile müssen mehrfach durchlaufen werden. Sie werden solange ausgeführt, wie eine bei jedem Schleifendurchlauf ausgewertete *Testbedingung* wahr (ungleich 0) ist. Beim Testergebnis falsch wird die Schleife verlassen. Der Schleifentest kann an verschiedenen Stellen stehen (Abb. 1). *Schleifenanweisungen* können Einzelanweisungen ⟨*Anweisung*⟩; oder ganze Anweisungsblöcke sein. Auch die Leeranweisung ; ist möglich.

**Schleifen mit abweisendem Test** `while(`⟨*Bedingung*⟩`)` ⟨*Anweisung*⟩`;`
prüfen vor jeder Ausführung der Schleifenanweisung, ob die Bedingung erfüllt ist. Ist die Bedingung schon beim ersten Test falsch, wird der Schleifenrumpf überhaupt nicht ausgeführt.

```
double wurzel(double x) // Wurzelziehen, Heron von Alexandria um 75 u.Z.
{
  double a = x, y = 1;  // Startbedingung x>=1
  while (x > y)
  {
    x = (x+y)/2;
    y = a/x;
  }
  return x;             // nach Schleifenende x=y=sqrt(a)
}
```

**Zählschleifen** `for(`⟨*Initialisierung*⟩`;` ⟨*Bedingung*⟩`;` ⟨*Inkrement*⟩`)` ⟨*Anweisung*⟩`;`
werden in anderen Sprachen so benannt, weil ein *Schleifenzähler* bei jedem Durchlauf um einen bestimmte Schrittweite (*Inkrement*) erhöht bzw. abgesenkt wird und der Endwert des Zählers bei Schleifenbeginn feststeht.

```
  for (fak = 1; n > 1; n--)  // n! = 1*2*...*n
    fak *= n;
```

Hier sind `for()`-Schleifen wesentlich flexibler einsetzbar. Sie haben eine gleichwertige Formulierung als `while()`-Schleife:

```
for (⟨Initialisierung⟩;
     ⟨Bedingung⟩;
     ⟨Inkrement⟩)
       ⟨Anweisung⟩;
```

```
{ ⟨Initialisierung⟩;
  while (⟨Bedingung⟩)
  { ⟨Anweisung⟩;
    ⟨Inkrement⟩;
  }
}
```

Für ⟨*Initialisierung*⟩, ⟨*Bedingung*⟩ und ⟨*Inkrement*⟩ dürfen beliebige Ausdrücke stehen. Zusammengesetzte, aber auch leere Ausdrücke sind möglich. Eine leere Bedingung gilt als wahr: `for(;;)` ⟨*Anweisung*⟩`;` wird damit zur *Endlosschleife*.

**Schleifen mit Test am Schleifenende** `do` ⟨*Anweisung*⟩`; while(`⟨*Bedingung*⟩`);` werden mindestens einmal durchlaufen, bevor die Bedingung getestet wird:

```
int eingabe_von_1_bis_5()
{ int num;
  do
    cin >> num;
  while (num < 1 || num > 5); // bei Fehleingabe wird wiederholt
  return num;
}
```

**Schleifenabbruch** kann mit `break;` innerhalb des Anweisungsblocks bewirkt werden. Die Schleife wird an dieser Stelle sofort verlassen (bei verschachtelten Schleifen nur die innerste). Eine Schleife mit Testbedingung in der Mitte:

```
for (;;)                          // Endlosschleife, auch while (true)
{ anw1;
  if (abbruchbedingung()) break;  // bei Abbruchbedingung verlassen
  anw2;
}
```

Durch `continue;` wird ans Blockende gesprungen, ohne die Schleife abzubrechen:

```
while ((p = naechster_passagier()) != niemand)
{ ausweiskontrolle(p);
  if (scheinbar_harmlos(p)) continue;
  zollkontrolle(p);
}
```

Die Anweisungen `break;` und `continue;` sollten als „verkapptes `goto`“ mit Vorsicht benutzt werden, da sie sich meist vermeiden lassen, ohne die Lesbarkeit zu gefährden:

```
while ((p = naechster_passagier()) != niemand)
{ ausweiskontrolle(p);
  if (!scheinbar_harmlos(p)) zollkontrolle(p);
}
```

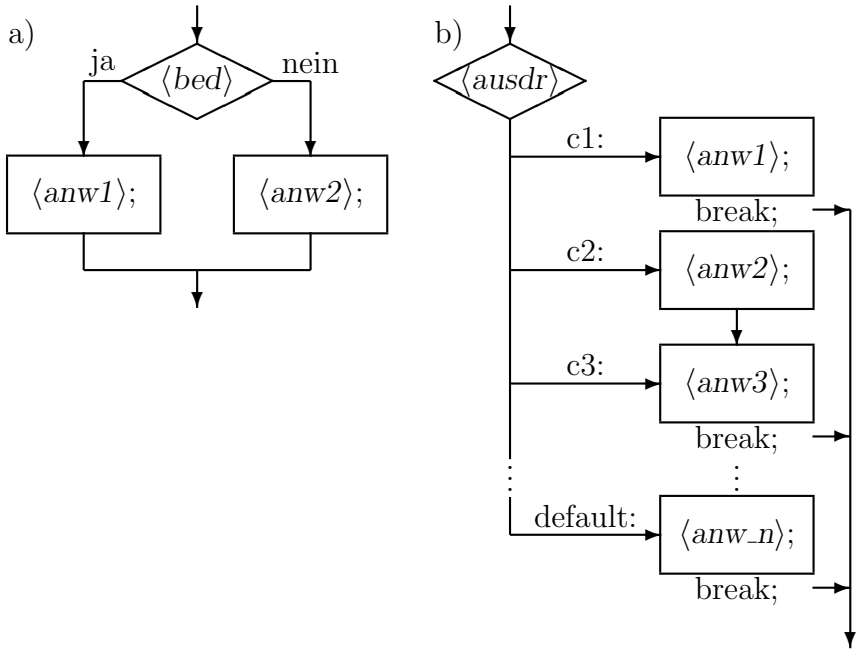



Abbildung 2: (a) Einfach- und (b) Mehrfachverzweigung.

### C.5.2 Verzweigungen

Manche Anweisungen sind nur unter bestimmten Bedingungen auszuführen (Abb. 2). Ab C++17 können Initialisierer angegeben werden.

**Entscheidungen** `if(`⟨*init;*⟩$_{opt}$⟨*Bedingung*⟩`)`⟨*Anweisung1*⟩`; else`$_{opt}$⟨*Anweisung2*⟩`;`$_{opt}$ erlauben die wahlweise Abarbeitung von Anweisungen:

```
if (gut(Erbse)) toepfchen++; // Aschenputtel's Tauben
else kroepfchen++;
```

Der `else`-Zweig muss, wenn vorhanden, unmittelbar der ⟨*Anweisung1*⟩ folgen. Bei verschachtelten Entscheidungen wird `else` immer an das unmittelbar vorhergehende `if()` gebunden. Eine andere Zuordnung ist durch Blockklammern `{ }` möglich. `if-else-if`-Leitern werden häufig verwendet:

```
if (⟨Bedingung1⟩)
    ⟨Anweisung1⟩; // nur eine Anweisung erfolgt
else if (⟨Bedingung2⟩)
    ⟨Anweisung2⟩;
...
else
    ⟨Anweisung n⟩; // diese, falls nichts zutrifft
```

**Mehrfachverzweigungen** lassen sich als `switch()`-Anweisung schreiben, wenn der Ausdruck der Verzweigungbedingung nur (wenige) ganzzahlige Werte annimmt:

```
switch (⟨init;⟩opt ⟨Ausdruck⟩)          // Falls ⟨Ausdruck⟩
{ case ⟨c_1⟩ :   ⟨Anweisungsfolge1⟩;    // einen Wert ⟨c_i⟩ annimmt,
                 break;                // wird ab entsprechender Stelle
  case ⟨c_2⟩ :   ⟨Anweisungsfolge2⟩;    // weitergearbeitet.
                 break;                // Bei break; endet Abarbeitung.
  ⋮
  case ⟨c_n⟩ :   ⟨Anweisungsfolge n⟩;
                 break;
  default :      ⟨sonstige Anweisungen⟩; // diese, falls nichts zutrifft
}
```

Jede Konstante ⟨*c_i*⟩ braucht ein eigenes *Label*. Es sind keine Bereichsangaben möglich.[26] Die Reihenfolge von Konstanten und `default:` ist frei wählbar. Mehrere `case`-Label können gemeinsamen Code besitzen, weil `break;` nicht nach jedem `case`-Zweig zu stehen braucht:

```
switch (ch = read_device())
{ case 1:
  case 2:
  case 3: flag=1;       // 1,2,3
          break;
  case 4: flag=2;       // nur 4
  case 5: error(flag);  // 4 und 5
          break;
  default: process(ch); // alle anderen ch
}
```

### C.5.3 Sprunganweisungen

Obwohl wegen der Gefahr von Spaghetti-Code verpönt[27], kann es nützlich sein, innerhalb einer Funktion an beliebiger Stelle eine Sprungmarke ⟨*Label*⟩`:` zu definieren, und mit `goto` ⟨*Label*⟩`;` an diese Stelle zu springen:

```
for (...;...;...)
{ for (...;...;...)
  { while (...)
    { ...
      if (katastrophe()) goto notfall;
      ...
    }
  }
}
...           // normales Ende
notfall: ... // retten, was zu retten ist
```

[26] Wie z. B. in der `case`-Anweisung von PASCAL.

[27] Siehe E. Dijkstra: *Goto statements considered harmful*, Comm. of the ACM, Bd. 11, März 1968.

## C.6 Funktionen

*Funktionen erlauben uns, auf dem aufzubauen, was andere vor uns programmiert haben.*
– Brian Kernighan & Dennis Ritchie

### C.6.1 Programm-Grundbausteine

Ein Programm ist im Wesentlichen eine Ansammlung von einander aufrufenden Funktionen. Funktionen erleichtern die Gliederung von Anweisungskomplexen in kleinere, auch wiederholt abarbeitbare Einheiten und verbergen Details vor dem Nutzer. Sie lassen sich in allgemein nutzbare *Bibliotheken* zusammenfassen und in mehreren Programmen verwenden.

### C.6.2 Vereinbarung

**Funktionsdeklarationen** ⟨*Ergebnistyp*⟩ ⟨*Funktionsname*⟩`(`⟨*Parameterliste*⟩`);`
erfolgen durch Angabe ihres *Funktionskopfes* (*Prototypen*), gefolgt von einem Semikolon. Eine Funktion kann mehrfach deklariert werden, sofern die Prototypen übereinstimmen. Eine Funktion muss vor dem Aufruf mindestens deklariert werden. Das kann auch noch
►C.6.5 innerhalb eines Anweisungsblocks geschehen, besser jedoch am Dateianfang:

```
double parabel(double x);
```

Die Parameterliste enthält, durch Komma getrennt, die *Parameter* der Funktion, jeweils mit Typ und, bei Deklarationen optional, Name.[28] Bei parameterlosen Funktionen kann die Parameterliste leer bleiben:

```
void faul(); // gleichbedeutend mit void faul(void);
```

Für variable Argumentlisten[29] werden drei Punkte ans Ende der Parameterliste gesetzt:

```
printf(const char* format, ...); // in <cstdio>
```

**Funktionsdefinitionen**
⟨*Ergebnistyp*⟩ ⟨*Funktionsname*⟩`(`⟨*Parameterliste*⟩`)` { ⟨*Anweisungen*⟩ }
geben nach dem Funktionskopf anstelle des Semikolons den *Funktionsrumpf* an. Der Funktionskopf muss bei getrennter Deklaration und Definition übereinstimmen.

```
double parabel(double x) { return x*x-2*x+4; }
```

Eine Funktion kann nur außerhalb anderer Funktionen und nur einmal definiert werden.[30] Alle Funktionen haben externe Bindung (sind global im Programm).[31] Die Reihenfolge der Funktionsdefinitionen im Quelltext ist beliebig.

[28] Dies führt zum *most vexing parse*, wenn eine Variablen-Initialisierung als Funktion aufgefasst wird: `int i(int(adouble));` — `i` als Funktion mit `int`-Parameter. Compiler geben im Unterschied zum Menschen dieser Lesart den Vorrang. Geschweifte statt runder Klammern für die Initialisierer lösen diese Mehrdeutigkeit auf: `int i{int(adouble)};`

[29] Zu ihrer Verarbeitung dienen die Makros aus `<cstdarg>`.

[30] Sprachen wie PASCAL erlauben Verschachtelungen (lokale Prozeduren).

[31] Als `static` oder in `namespace{...}` deklarierte Funktionen sind nur innerhalb einer Datei sichtbar.

**Die Rückkehranweisung** `return` ⟨*Ausdruck*⟩`;`
beendet die Abarbeitung der Funktion. Der Wert von ⟨*Ausdruck*⟩ wird an den Aufrufer der Funktion übergeben und kann, muss aber nicht, von diesem ausgewertet werden:

```
double y = 2 * parabel(2.7183);
```

Bei Funktionen vom Typ `void` kann die Anweisung `return;` ohne Ausdruck entfallen. Dann endet die Abarbeitung des Funktionsrumpfes bei der schließenden Klammer `}`. Rückkehranweisungen können mehrfach, auch mitten im Funktionsrumpf stehen:

```
long ggT(long a, long b)    // groesster gemeinsamer Teiler (a>=0, b>0)
{                           // Algorithmus von Euklid  (365?-300? v.u.Z.)
  if (a == 0) return b;
  return a>b ? ggT(a%b, b) : ggT(b%a, a); // rekursive Funktion
}
```

**inline** deklarierte kleine Funktionen vermeiden uneffiziente Funktionsaufrufe. Statt dessen wird der Rumpf sinngemäß in den Code eingefügt. Sie sind typsicher und haben nicht die von Präprozessormakros her bekannten und berüchtigten Nebeneffekte.

```
inline int minimum(int x, int y) { return x<y ? x : y; }
// ...
{ int m = 3;
  int n = minimum(--m, 3); // n=2
}
```

**Überladene Funktionen** gleichen Namens, aber unterschiedlicher Parametertypen, sind definierbar. Der Compiler prüft beim Aufruf anhand der Parametertypen, welche Funktion aufgerufen werden soll:

```
char  minimum(char  x, char  y) { return x<y ? x : y; }
float minimum(float x, float y) { return x<y ? x : y; }

char   a = 2, b = 3;
float f1 = 1, f2 = 2;
char   c = minimum(a, b);   // minimum(char, char)
float f3 = minimum(f1, f2); // minimum(float, float)
```

**Funktionsschablonen** (engl. `template`) legen Parametertypen erst beim Funktionsaufruf fest:

```
template <typename T>
T minimum(T x, T y) { return x<y ? x : y; }

int i = 2, j = 3;
double d1 = 1, d2 = 2;
int     k = minimum(i, j);   // minimum(int, int)
double d3 = minimum(d1, d2); // minimum(double, double)
```

### C.6.3 Parameter

**Wertparameter** erhalten beim Aufruf einer Funktion eine Kopie der Werte der aktuellen Parameter. Die Werte sind innerhalb der Funktion frei änderbar, ohne Auswirkungen auf übergebene äußere Variable (*Kopiersemantik*):

```
int digits(long num)  // Anzahl der Ziffern
{
  int dig = 1;
  while (num /= 10) dig++;
  return dig;
}
```

Beim Aufruf `n = digits(MAXINT);` wird der Wert von `MAXINT` nicht verändert.

**Standardparameter** setzen automatisch Vorgabewerte ein, wenn beim Funktionsaufruf Parameter (von rechts nach links) weggelassen werden:

```
float dezimal(int zl=0, int nn=1)   // Dezimalwert eines Bruchs
{                                   // nn != 0
  return float(zl)/nn;
}

float null = dezimal(), zehn = dezimal(10), ein_halb = dezimal(1,2);
```

**Referenzparameter** können die Werte aufrufender Variablen verändern (*Referenzsemantik*). Sie führen nur ein neuen Namen (*Alias*) für sie ein:

```
void swap(int& x, int& y) { int t = x; x = y; y = t; }
// ...
int i = 2, j = 3;
swap(i, j);    // i = 3; j = 2;
```

In C sind Referenzparameter nur indirekt über Zeiger möglich:

```
void swap(int *x, int *y) { int t = *x; *x = *y; *y = t; }
// ...
int i = 2, j = 3;
swap(&i, &j);  // i = 3; j = 2;
```

**Rechtswertreferenzen** (*rvalue references*) `&&` als Parameter bzw. Rückgabewerte erlauben *Verschiebesemantik* ohne teure Kopien, sofern der Typ `T` diese unterstützt:

```
template <typename T>
T&& move(T&& rvalue_ref) { return rvalue_ref; }

template <typename T>
void swap(T& x, T& y) { T t = move(x); x = move(y); y = move(t); }
```

**Felder als Parameter** werden nicht als Ganzes kopiert, sondern nur der Zeiger auf das Anfangselement übergeben (*pointer decay*). Größenangaben haben lediglich Informationswert für den Nutzer:

```
int f1(int a[10]);
int f2(int a[]);   /* dasselbe */
int f3(int *a);    /* meist so */
```

### C.6.4 Hauptprogramm

Eine Funktion

```
int main(int argc, char *argv[])
{
  // ... Anweisungen
  return statuscode;
}
```

muss in jedem Programm enthalten sein. In dieser Funktion beginnt der Programmablauf und hier endet er mit der Rückgabe eines Statuscodes an das aufrufende Programm (Betriebssystem).[32] Ein Programm, das `0` zurückliefert, gilt als erfolgreich beendet. Die Anweisung `return 0;` am Ende von `main()` kann entfallen. Die Parameter `argc` und `argv` enthalten Anzahl und Zeichenketten der Kommandozeilenparameter:[33]

- `argv[0]` ist stets der Programmname des Aufrufes.
- `argv[argc-1]` ist der letzte gültige Parameter.

Programme können als `int main()` deklariert werden, wenn sie keine Kommandozeilenparameter auswerten.

### C.6.5 Deklarationsdateien

Wird eine Funktion in mehreren Quelldateien verwendet, muss ihr Prototyp in allen Dateien übereinstimmen. Erleichtert wird das, indem die Deklarationen global nutzbarer (aus Modulen *exportierter*) Funktionen in *Deklarationsdateien* (*Header*) untergebracht werden.[34] Vor der Verwendung der Funktion wird die zugehörige Deklarationsdatei mit

▶C.8.2

```
#include <header1>     // Datei im Standard-Include-Verzeichnis
#include "header2.h"   // Datei im aktuellen Verzeichis
```

eingebunden.

▶C.7.5

[32] Mit Compilerpragmas lassen sich auch Funktionen vom Typ `void f(void)` vor bzw. nach `main()` in vorgegebener Rangfolge ausführen:
`#pragma startup` ⟨*funktion1*⟩ ⟨*rang1*⟩
`#pragma atexit` ⟨*funktion2*⟩ ⟨*rang2*⟩
Das ist nur selten nötig und besser, weil portabel, mit Klassen realisierbar.

[33] In einigen Umgebungen sind weitere Parameter möglich.

[34] Üblich sind Dateinamen mit `*.h` als Endung. Standardheader müssen nicht einmal Dateien sein.

## C.7 Vorverarbeitung

Quelltextzeilen, die das Doppelkreuz # als erstes (Nicht-whitespace-) Zeichen enthalten,
►C.8.1 gelten als *Präprozessoranweisungen*. Mit dem Backslash \ am Zeilenende lassen sich diese auch auf Folgezeilen ausdehnen.

Die Vorverarbeitung umfasst das Entfernen von Kommentaren und überflüssigen Leerzeichen, den Textersatz (das Abarbeiten von *Makros*), das Einbinden von anderen Quelltexten, Entscheidungen zur bedingten Übersetzung von Quellteilen und das Einfügen des aktuellen Quelldateinamens und der aktuellen Quelltextzeile (für Fehlermeldungen des Compilers). Erst deren Ergebnis[35] wird dem Compiler zur Übersetzung vorgelegt.

### C.7.1 Makrokonstanten und -funktionen

Durch `#define`-Anweisungen werden Präprozessorkonstanten und -funktionen (*Makros*) definiert. Deren Namen[36] werden nur dem Präprozessor bekannt.

**Makrokonstanten** werden mit der Zeichenfolge bis zum Zeilenende verbunden:

```
#define PI 3.1415
#define DWORD unsigned long
#define SHORT
```

**Makrofunktionen** haben mindestens einen Parameter:

```
#define SQR(x)   ((x)*(x))
#define Ich_bin_nicht_da(weg)
```

Sobald ein definierter Makroname im Quelltext (außerhalb von Zeichenketten) verwendet wird, wird dieser durch den Text des Makros ersetzt.[37] Für die Parameter kann beim Aufruf beliebiger Text stehen. Dieser Text wird sooft kopiert, wie der Parametername in der Makrodefinition erscheint. „Leere“ Makrokonstanten bzw. -funktionen werden einfach aus dem Quelltext entfernt; auch deren aktuelle Parameter verschwinden:

```
DWORD u=2*PI;
SHORT int i=SQR(123);
Ich_bin_nicht_da(haha);
```

werden für den Compiler zu

```
unsigned long u=2*3.1415;
int i=((123)*(123));
;
```

Makrodefinitionen lassen sich mit `#undef` rückgängig machen, um die Wirkung des Makros lokal zu begrenzen oder um sicherzugehen, dass es sich bei Funktionsaufrufen wirklich um Funktionen handelt:

```
#undef SQR(x)
```

[35] Manche Systeme speichern unter bestimmten Bedingungen `*.i`-Zwischendateien (`i` wie *input*).
[36] Makronamen werden meist großgeschrieben. Viele vordefinierte beginnen mit Unterstrichen.
[37] Das kann seitenweise Tipparbeit ersparen (auf Kosten der Codegröße).

### C.7.2 Fehler und Gefahren

Der Präprozessor kann weder C noch C++. Deshalb sollten Makrorümpfe mit Operatoren und Parameter geklammert werden, mehrfach verwendete Makroargumente keine Seiteneffekte erzeugen. Inkrement, Dekrement, zählende Funktionen u. ä. sind dort zu meiden. Makrorümpfe sollten zusammenhängend sein. Ausdrücke sind besser als Anweisungen, eine Anweisung ist besser als mehrere. Hier einige abschreckende Beispiele:

```
#define succ(x) x+1
#define sqr(x) x*x
int i = succ(5)/2; // i = 5+1/2;   ergibt 5 statt 3
int j = sqr(1+2);  // j = 1+2*1+2; ergibt 5 statt 9
int k = 2;         // nicht definiertes Verhalten:
int n = sqr(k++);  // n = k++*k++; also n=4, 6 oder 9; k=4 statt k=3

#define PRINTLN(fmt,x) printf(fmt,x); putchar('\n')
for(i=0; i<10; i++) PRINTLN("%d",x); // nur ein Zeilenvorschub
```

Verstümmelungen bis zur Unkenntlichkeit sind möglich:

```
#include <iostream>
#define PROGRAM int main()
#define VAR
#define BEGIN {
#define END }
#define REAL float
#define PI 3.1415297
#define READLN(x)  std::cin >> x
#define WRITELN(x) std::cout << x << '\n'
#define IF if(
#define THEN )
#define ELSE else
#define REPEAT do{
#define UNTIL(x) } while(!(x))

// In welcher Sprache ist das geschrieben?
PROGRAM
BEGIN
  VAR REAL r,area;
  REPEAT
    READLN(r);
    area=PI*r*r;
    IF area > r THEN BEGIN WRITELN(area); r=r/2; END
    ELSE r=r*2;
    WRITELN(r);
  UNTIL (r<0.1);
END
```

### C.7.3 Zeichenkettenmanipulation

**Umwandeln von Parametern in Zeichenketten** wird durch ein Doppelkreuz unmittelbar vor dem Makroparameter in der Definition erreicht. Dies ist u. a. nützlich zur Fehlersuche:

```
#include <iostream> // nur in C++ verwendbar, aber typsicher
#define TRACELN(x) std::cout << #x << " = " << (x) << '\n'

TRACELN(1+2*3-4%5); // erzeugt Ausgabe: 1+2*3-4%5 = 3
```

**Doppelte Doppelkreuze** im Makro ziehen die beiden Argumente zu einem Namen zusammen, der dann vom Präprozessor und vom Compiler als Einheit (ein Token) behandelt wird:[38]

```
#define DECLARE(a,b) a##b
DECLARE(STACK,int) a; // wird STACKint a;
```

### C.7.4 Vordefinierte Makros

Mit (mehreren) Unterstrichen versehene Makros sind schon vordefiniert. Informationen über die übersetzte Datei liefern die Bezeichner:

```
__DATE__ __TIME__ __FILE__ __LINE__ // Datei-Informationen
__func__                            // Funktionsname lokal
__CDECL__ __PASCAL__                // Linker-Optionen
__STDC__ __cplusplus                // ANSI C, C++ (Version)
```

Das Betriebssystem bzw. der Compiler fügen (implementationsspezifisch) weitere hinzu:

```
__WIN32__ _MSC_VER
__GNUC__  MINGW32
```

Manche Makros lassen sich durch Präprozessoranweisungen beeinflussen:

```
#line 110 "hier.txt"
```

gaukelt dem Compiler vor, in Zeile 110 der Datei `hier.txt` zu sein. Der Dateiname ist optional.

```
#pragma ....
```

enthalten implementationsspezifische Anweisungen an den Compiler.

```
#error Das ist ein Fehler
```

stoppt die Übersetzung mit dem angegebenen Fehlertext.

[38] In frühen C++-Versionen wurde diese Technik als Ersatz für Klassen-Templates genutzt.

### C.7.5 Einlesen von Dateien

In Quelltexte können weitere Teildateien eingebunden werden. Große Quellen lassen sich so in aufeinanderfolgende Teile gliedern bzw. ein Teil in mehrere Quelltexte einbauen:

```
#include <datei2>
#include "datei1"
```

Enthält der Dateiname der Teildatei nicht den vollständigen Pfad, beginnt die Suche im aktuellen Verzeichnis und wird bei Misserfolg im Standard-Include-Verzeichnis des Compilers fortgesetzt. Bei Angabe des Dateinamens in spitzen Klammern wird nur im Include-Verzeichnis gesucht.

In mehreren Quelltexten genutzte Funktionsnamen, Makros, oder globale (externe) Variablen werden in Deklarationsdateien `*.h` zusammengefasst. Diese sollten nur einmal eingelesen werden, da doppelte Deklarationen einen Fehler darstellen (können). Um zu entscheiden, ob die Datei schon gelesen wurde, haben die Header meist einen Rahmen, der *bedingte Übersetzung* erzwingt:

```
#ifndef MYHEADER_H
#define MYHEADER_H
  /* Deklarationen hier */
#endif
```

### C.7.6 Bedingte Übersetzung

Durch Einfügen von `#if`-Anweisungen lässt sich steuern, welche folgenden Quelltextzeilen zum Programm gehören sollen. Hinter `#if` kann ein beliebiger konstanter Ausdruck stehen, der auch (vorher definierte) Makrokonstanten enthalten kann:

```
#if defined(__cplusplus)
  /* C++ - Definitionen  */
#elif defined(__STDC__)
  /* ANSI C Definitionen */
#else
  #error Sprache nicht unterstuetzt
#endif
```

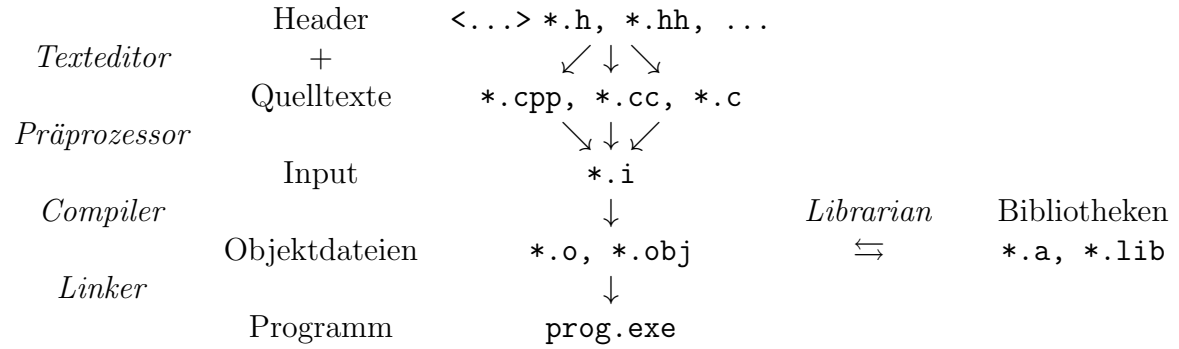
Die entstehende `#if-#else`-Leiter muss immer mit `#endif` abgeschlossen werden. `#elif`-Zweige und `#else`-Zweig sind optional. Zur Abfrage der Makrodefinitionen existieren die Abkürzungen

```
#ifdef MAKRO          /* #if defined(MAKRO)  */
#ifndef MAKRO         /* #if !defined(MAKRO) */
```

## C.8 Programmorganisation

### C.8.1 Übersetzungsprozess

Zu einem Programm können mehrere Quelltextdateien gehören. Bei ihrer Übersetzung entstehende *Objektdateien* können mit Objektdateien aus anderen Programmiersprachen und *Bibliotheken* zum ausführbaren Programm (*Binärdatei*) zusammengeführt werden:




Bibliotheken wie `libm.a` oder `math.lib` sind Sammlungen von Objektdateien[39], aus denen benötigte Teile zum Programm hinzugefügt (*gelinkt*) werden. Übersetzungsprozess und Projektverwaltung der Compiler ist systemabhängig unterschiedlich organisiert. Dies kann manuell mit Kommandozeilenbefehlen

```
g++ hello.cpp -o hello
```

geschehen oder über `make`-Dateien, wobei nur geänderte Teile neu übersetzt werden müssen. Andere Systeme integrieren Projektverwaltung, Compiler und Linker in eine Entwicklungs-Umgebung.

### C.8.2 Modularisierung

Modularisierung ist eine Technik zum *Aufbrechen der Komplexität* von Programmieraufgaben. Inhaltlich zusammengehörende Funktionen und Variablen lassen sich als *Modul* (Übersetzungseinheit) in einer Quelltextdatei `name.cpp` bündeln. Das Modul kann dann als Objektdatei `name.o` oder Bibliothek `libname.a` in mehrere Programme eingebunden werden, ohne geändert oder neu übersetzt werden zu müssen.

Die Kommunikation mit dem Modul erfolgt nur über die festgelegte Schnittstelle: die zugehörige Deklarationsdatei `name.h`. Als `extern "C"` deklarierte Funktionen können auch in C-Programme eingebunden werden. Zur Verwendung des Moduls ist keine Kenntnis der inneren Funktion erforderlich. Schnittstellendeklaration und -dokumentation (sollten) genügen. Module unterstützen damit das *Geheimnisprinzip*. Hilfsfunktionen und Hilfsvariablen lassen sich als in unbenannten *Namensräumen* deklarieren.[40] Damit sind sie vor dem Zugriff aus anderen Programmteilen geschützt und Kollisionen mit gleichlautenden Bezeichnern anderer Module nicht möglich.

►C.9 Das Klassenkonzept geht noch einen Schritt weiter, indem jede Variable zu einem (unabhängigen) Modul mit eigenen Daten wird.

[39] Dateiendungen sind plattformspezifisch.

[40] Vor der Erfindung der Namensräume wurden solche Namen als `static` deklariert. Globale Variablen stellen (meist) ein Sicherheitsrisiko dar, da sie über `extern`-Deklarationen manipulierbar sind.

### C.8.3 Namensräume

**Namenskonflikte** können die Integration von Modulen, meist unterschiedlicher Herkunft, verhindern, wenn derselbe Name in ihnen unterschiedlich definiert wurde:

```
int init() { /* tue dies */  } // in Modul A
int init() { /* tue jenes */ } // in Modul B
```

Manchmal wird das erst spät bemerkt:

```
#include "headerA.h"  // int init();
#include "headerB.h"  // int init();
int i = init();       // welches init() ? => Linker-Fehler
```

**Namensräume** `namespace` ⟨*Name*⟩$_{opt}$ `{` ⟨*Deklarationen, Definitionen*⟩ `}`
begrenzen die globale „Umweltverschmutzung“:

```
namespace Modul_A {
  int init() { /* tue dies */  }
}
namespace Modul_B {
  int init() { /* tue jenes */ }
}
```

Beim Aufruf muss dann der Namensbereich mit angegeben werden (*Bereichsauflösung*):

```
#include "headerA.h"     // int init();
#include "headerB.h"     // int init();

int i = Modul_A::init(); // o.k.
```

Einzelne Namen oder ganze Namensbereiche können in den aktuellen Block (oder global) importiert werden, solange dadurch kein Konflikt entsteht:

```
#include "headerA.h"     // int init();
#include "headerB.h"     // int init();

using Modul_A::init;     // einzelner Name
using namespace Modul_A; // ganzer Bereich
int i = init();          // Modul_A::init
int j = Modul_B::init(); // o.k.
```

Ein Namensbereich darf mehrmals geöffnet und geschlossen werden:

```
namespace Modul_A {
 struct Bruch { int z, n; };
 int nenner(Bruch b) { return b.n; }
}
```

Tabelle 7: Header der C++-Standardbibliothek und ihr Einsatzgebiet (Auswahl).

| | | |
|---|---|---|
| `<cstddef>`* | `<cstdlib>`* | Grundlagen der C-Standardbibliothek |
| `<typeindex>` `<type_traits>` | `<typeinfo>` | Laufzeit-Typinformation |
| `<limits>` | | Wertebereichsinformation |
| `<cfenv>`* | | Steuerung Gleitkommaeinheit |
| `<memory>` | `<new>` | Freispeicherverwaltung |
| `<cassert>`* `<exception>` `<system_error>` | `<cerrno>`* `<stdexcept>` | Fehlerbehandlung |

In einem (geschachtelten) Namensbereich ist der Zugriff auf alle vorher im gleichen und in übergeordneten Namensbereichen deklarierten und importierten Namen erlaubt:

```
namespace std::literals {
  inline namespace string_literals {
    string operator "" s (const char* str, size len);
  }                       // "Hallo"s wird zu einem std::string
}
```

Deklarationen eines `inline`-Namensbereiches sind auch im umgebenden sichtbar. Funktionen, deren Argumenttypen in demselben Namensbereich definiert wurden, können ohne Nennung des Namensbereiches aufgerufen werden (*argument dependent/Koenig lookup*):

```
Modul_A::Bruch b = {4, 6};
int n = nenner(b);
```

**Unbenannte Namensräume** kapseln globale Namen, die nur in einer Übersetzungseinheit verfügbar sein sollen (`static` file scope):

```
namespace { int x = 1; } // x nur in dieser Datei
int y = x;
```

**Header der Standardbibliothek** (Tab. 7 und Tab. 8) platzieren ihre Deklarationen im Namensraum `std`. C++ erbte einen kleinen Teil seiner Bibliothek von C (durch * markiert).

Tabelle 8: Header der C++-Standardbibliothek (Fortsetzung).

| | | |
|---|---|---|
| `<chrono>` | `<ctime>`* | Zeitfunktionen |
| `<cctype>`* | `<cwctype>`* | Zeichenarten |
| `<string>` | `<string_view>` | Zeichenketten |
| `<iostream>` | | Ein- und Ausgabeströme |
| `<iomanip>` | | Formatierung |
| `<fstream>` | `<sstream>` | Datei-/Zeichenkettenströme |
| `<filesystem>` | | Dateisystem |
| `<locale>` | | Internationalisierung |
| `<regex>` | | reguläre Ausdrücke |
| `<complex>` | | komplexe Zahlen |
| `<cmath>`* | | mathematische Funktionen |
| `<random>` | | Zufallszahlgeneratoren |
| `<ratio>` | | Verhältniszahlen |
| `<tuple>` | | Tupel |
| `<any>` | | Wert eines beliebigen Typs |
| `<optional>` | | evtl. vorhandener Wert |
| `<variant>` | | typsicherer Ersatz für `union` |
| `<bitset>` | | Bitmengen |
| `<initializer_list>` | | Initialisierer-Listen |
| `<valarray>` | | Zahlenfelder |
| `<array>` | `<deque>` | |
| `<forward_list>` | `<list>` | |
| `<map>` | `<set>` | Datencontainer |
| `<unordered_map>` | `<unordered_set>` | |
| `<vector>` | | |
| `<queue>` | `<stack>` | Warteschlangen, Stapel |
| `<functional>` | | Funktionsadapter |
| `<iterator>` | | Iteratoren |
| `<utility>` | | Hilfsfunktionen der Containerbibliothek |
| `<algorithm>` | `<numeric>` | Algorithmen |
| `<atomic>` | | |
| `<condition_variable>` | | |
| `<future>` | | nebenläufige Prozesse |
| `<mutex>` | `<shared_mutex>` | |
| `<thread>` | | |

## C.9 Klassenkonzept

### C.9.1 Objektbasierte Programmierung

Objekte (*Instanzen*) einer *Klasse* weisen bestimmte gemeinsame Merkmale (*Komponenten*) auf und verhalten sich gleichartig. Sie verfügen über eigene Funktionen (*Methoden*) zur Bearbeitung ihrer eigenen Daten (*Attribute*).

**Die Klassendefinition** `class` ⟨*Klassenname*⟩ { ⟨*Komponenten*⟩ } ⟨*Instanzenliste*⟩`;`
(öffentliche Schnittstelle) ist eine Typvereinbarung, die Daten und Methoden dieses Typs
►S. 44 zusammenfasst. Bis auf die voreingestellten Zugriffsrechte sind `struct` und `class` gleichwertig. Der ⟨*Klassenname*⟩ ist Typname und zugleich Namensbereich.

```
class Stack {          // oder struct
public:
  Stack();             // Konstruktor
  void push(T x);      // Methoden
  T pop();
  int empty() const;
  int full()  const;
private:
  static const int size = 100;
  T a[size];           // Datenkomponenten
  int pos;
};                     // noch keine Instanz
```

**Instanzdeklarationen** ⟨*Klassenname*⟩ ⟨*Instanz*⟩`,` `*`⟨*Instanzzeiger*⟩`;`
vereinbaren Variablen vom Typ ⟨*Klassenname*⟩. Komponenten (Daten und Methoden) von Instanzen werden von außerhalb der Klasse als ⟨*Instanz*⟩`.`⟨*Komponente*⟩ bzw. ⟨*Instanzzeiger*⟩`->`⟨*Komponente*⟩ angesprochen:

```
void demo(Stack& s, T& x)
{
  if (!s.full() ) s.push(x);
  if (!s.empty()) x=s.pop();
}
```

**Die Implementation** der Methoden (Definition)
⟨*Ergebnistyp*⟩ ⟨*Klassenname*⟩`::`⟨*Methode*⟩`(` ⟨*Parameterliste*⟩ `)` { ⟨*Methodenrumpf*⟩ }
kann außerhalb des Klassenrumpfes, auch für Nutzer unzugänglich, in einer getrennten Datei erfolgen. Die Methode hat Zugriff auf alle Komponenten ihrer Klasse.

Kleine Funktionen können `inline` deklariert werden. Dann muss der Quelltext der Implementierung allerdings im Header zugänglich bleiben. Innerhalb der Klassendefinition aufgeführte Methodenrümpfe gelten automatisch als `inline` deklariert.

Als `const` markierte Methoden dürfen keine Daten verändern, sondern nur lesend auf diese zugreifen, und nur konstante (lesende) Methoden aufrufen. Anders herum dürfen bei konstanten Instanzen auch nur konstante Methoden genutzt werden.

```
class Stack {
  // ...
  int empty() const { return pos == 0; }
  int full()  const { return pos == size; }
};
inline void Stack::push(T x) { a[pos++] = x; }
inline T Stack::pop() { return a[--pos]; }
```

**Konstruktoren** ⟨*Klassenname*⟩`(`⟨*Parameterliste*⟩`)` und der parameterlose

**Destruktor** `~`⟨*Klassenname*⟩`()` erledigen Routineaufgaben beim Erschaffen und Vernichten einer Instanz wie die Anfangswertbelegung der Komponenten (*Initialisiererliste*):

```
Stack::Stack() : pos{0} {}
```

Eine Klasse kann mehrere Konstruktoren besitzen, jedoch nur einen Destruktor. Der Compiler erzeugt für jede Struktur die speziellen Methoden:[41] ▶S. 47

```
struct Empty {
  Empty() {}                                      // Standardkonstruktor
  Empty(const Empty& X) {}                        // Kopierkonstruktor
  Empty(Empty&& X) {}                             // Verschiebekonstruktor
  Empty& operator=(const Empty& X){return *this;}// Zuweisungsoperator
  Empty& operator=(Empty&& X) { return *this; }  // Verschiebezuweisung
  ~Empty() {}                                     // Destruktor
};
Empty a;               // Standardkonstruktor
Empty b{a};            // Kopierkonstruktor
Empty c{std::move(a)}; // Verschiebekonstruktor, a nicht mehr nutzbar
b = c;                 // Zuweisungsoperator
b = std::move(c);      // Verschiebezuweisung, c nicht mehr nutzbar
```

Kopie und Zuweisung sorgen für elementweise Übernahme der Attribute. Verschiebekonstruktor und -zuweisung können verschiebbare Daten billig aus dem rechtsseitigen Operanden holen. Sind eigene Konstruktoren definiert, muss einer aufgerufen werden:

```
struct X {
  X(char c) : X(int(c)) {} // delegiert Aufgabe
  explicit X(int i) {}     // explicit gegen versehentliche Typumwandlung
  X() = default;           // trotzdem erzeugen
  X(const X&) = delete;    // keine Kopie erlaubt
};
X x{1}; // Konstruktor X(int)
x = 3;  // Fehler: impliziter Aufruf x = X{3}
X z[5]; // Felder nutzen Standardkonstruktor
```

[41] Werden eigene Konstruktoren/Zuweisungen/Destruktor definiert, erzeugt der Compiler einige der vorgefertigten Methoden nicht mehr (Tab. 24 auf S. 131). Das reservierte Wort `this` in Methoden ist ein Zeiger auf die aufrufende Instanz.

**Zugriffsrechte** auf ihre Komponenten (Daten und Methoden) werden von den Klassen verliehen und bieten Schutz gegen unbefugte Manipulationen. Während bei `struct` auf alle Komponenten *öffentlich* zugegriffen werden kann, sind `class`-Komponenten nur innerhalb der Klasse ansprechbar (*privat*).

Nach `public:` deklarierte Bestandteile können von außerhalb angesprochen werden.
▶C.9.4 `protected:`-Abschnitte erlauben *geschützten* Zugriff nur für die Klasse und ihre Erben. `private:`-Komponenten sind auch den Nachkommen nicht zugänglich. In der Klassendefinition können Abschnitte mit Zugriffsrechten in beliebiger Reihenfolge, beliebig oft aufgeführt werden.

Eine Klasse kann andere Klassen und Funktionen zu *Freunden* (`friend`) erklären, die dann Zugriff auf private Daten und Methoden haben.[42]

```
class X {
public:
  X(int a) : x{a} {}
  friend sum(X xx, int i);
protected:
  int add(int i) { return x + i; }
private:
  int x;
} a(10);

int sum(X xx, int i) { return xx.x+i; } // Freund von X

int j = sum(a,2); // j = 12
int k = a.add(2); // Fehler: X::add() nicht public
```

**Statische Klassenkomponenten** können mit ⟨*Klassenname*⟩`::`⟨*Komponente*⟩ oder mit ⟨*Instanz*⟩`.`⟨*Komponente*⟩ aufgerufen werden. Als `static` deklarierte Methoden haben nur Zugriff auf statische Klassendaten. Solche Daten sind nur einmal pro Klasse vorhanden und müssen `inline` oder außerhalb der Klasse als globale Variable definiert und gelinkt werden:

```
class Count {
  inline static int cnt = 0; // oder: static int cnt; // (vor C++17)
public:
  Count()  { ++cnt; }
  ~Count() { cnt--; }
  static int instances() { return cnt; }
};

// int Count::cnt = 0; // einmal im Programm, wenn nicht inline
```

[42] Private Konstruktoren erlauben das Schaffen einer Instanz dieser Klasse nur über Freunde oder statische *Fabrikmethoden*. Von Klassen mit geschützten Konstruktoren können Erben erzeugt werden.

### C.9.2 Generische Programmierung

**Klassenschablonen** `template<`⟨*Parameterliste*⟩`>` ⟨*Schablone*⟩`;`
definieren Entwürfe von Klassen, die sich nur durch Konstanten oder in den Datentypen von bestimmten Komponenten, Methodenparametern und Rückgabewerten unterscheiden, welche in der Parameterliste aufgeführt sind:

```
template <class T, int size>
class Stack {
public:
  void push(T x) { a[pos++] = x; }
  T pop() { return a[--pos]; }
  int empty() const { return pos == 0; }
  int full()  const ;
private:
  T a[size];
  int pos = 0;
};
```

Außerhalb der Schablone definierte Methoden werden mit `template<`⟨*Parameterliste*⟩`>` eingeleitet. Hinter dem Namen der Klassenschablone sind die Argumente der Parameterliste in spitzen Klammern aufzuführen. *Spezialisierungen* erlauben Sonderbehandlungen bei ausgewählten Parametern. Der gesamte Quelltext muss vor der Nutzung definiert sein. Die Implementation von Klassenschablonen kann nicht versteckt werden.

```
template <class T, int size>
int Stack<T, size>::full() const { return pos == size; }

template <>                          // Extrawurst
int Stack<int,100>::full() const { std::cerr<<'\a'; return pos == size; }
```

**Wiederverwendung** einmal geschriebener Schablonen erfolgt durch die Konkretisierung der Schablonenparameter bei Nutzung einer solchen Klasse. Nach Festlegung aller Typen und Konstanten wird Code erzeugt. Schablonen sind auch ineinander schachtelbar.

```
typedef Stack<int, 100> IntStack;  // Stack, 100 int's
#include<vector>                   // vector von Stacks mit 256 double's
typedef std::vector<Stack<double, 256>> StackArray;
```

Die Nutzung ihrer Instanzen unterscheidet sich nicht von anderen Klassen:

```
int demo(int x, IntStack& s, StackArray& array)
{ if (!s.full() ) s.push(x);
  if (!s.empty()) x = s.pop();
  array[8].push(1.2345); // auf achten Stapel schieben
  return x;
}
```

### C.9.3 Überladen von Operatoren

Für benutzerdefinierte Typen (Klassen und Strukturen) lassen sich Operatoren neu definieren. Ein Operator ⟨*op*⟩ wird als Funktion ⟨*Typ*⟩ `operator` ⟨*op*⟩(⟨*Parameterliste*⟩) mit einer neuen Bedeutung *überladen*. Mathematische Strukturen erhalten dadurch ihre gewohnte Operator-Schreibweise.

```
struct Bruch {
  long z,n;
  Bruch(long z=0, long n=1) : z{z}, n{n} { kuerzen(); }

  // Rechenoperationen
  Bruch& operator+=(const Bruch& b);
  // ...
private:
  void kuerzen(); // z,n teilerfremd machen
};
```

**Operatormethoden** erhalten den ersten Operanden implizit (`*this`). Das rechte Argument zweistelliger Operationen wird der Methode als Funktionsparameter übergeben:

```
Bruch& Bruch::operator+=(const Bruch& b)
{
  return *this = Bruch{z*b.n + n*b.z, n * b.n};
}

Bruch a{1,2}, b{2,3}, c{3,4};
a += b;      // bedeutet:  a.operator+=(b);
```

**Verbundzuweisungen** können meist die zugehörigen Binäroperatoren elegant und effizient implementieren. Das erlaubt schlanke Schnittstellen bei weniger Schreibarbeit.

**Globale Operatorfunktionen** können von einer Klasse als `friend` deklariert werden, um Zugriff auf private Klassenbereiche zu ermöglichen, müssen jedoch nicht. Globale zweistellige Operatoren arbeiten gewöhnlich auf gleichberechtigten linken und rechten Argumenten. Damit sind auch implizite Konversionen des linken Operanden möglich.

```
Bruch operator+(const Bruch& a, const Bruch& b)
{
  return Bruch{a} += b;
}

c = a+b;     // bedeutet: c=operator+(a,b);
c = a+1;     //           c = a + Bruch{1};
c = 1+a;     //           c = Bruch{1} + a;
```

**Ein- und Ausgabeoperatoren** `>> <<` für nutzerdefinierte Typen sind als globale Funktionen zu definieren, deren Rückgabewert und linker Parameter eine `std::istream`- bzw. `std::ostream`-Referenz ist. Bei einem Fehlschlag des Lesens sollte das rechtsseitige Argument seinen alten Wert behalten:

```
#include <iostream>

std::ostream& operator<<(std::ostream& os, const Bruch& b)
{
  return os << b.z << '/' << b.n;
}

std::istream& operator>>(std::istream& is, Bruch& b)
{
  char c = ' ';
  int z, n = 1;
  if (is >> z >> c >> n && c == '/') b = Bruch{z, n};
  return is;
}

void in_out(Bruch a, Bruch& b)
{
  std::cout << a << "\nEingabe Bruch: ";
  std::cin  >> b;
}
```

**Zuweisungsoperator** `=` und Kopierkonstruktor, evtl. auch Verschiebekonstruktor bzw. -zuweisung, sollten umdefiniert werden, wenn eine Klasse Ressourcen verwaltet, die eines Destruktors bedürfen (z.B. dynamische Zeiger: Referenzproblem bei bitweiser Kopie).[43]

```
class X {
public:
  X() = default;
  X(T t) : p{new T(t)} {}
  X(const X& x) : p{new T(*x.p)} {}  // Kopierkonstruktor
  X(X&& x) { std::swap(p, x.p); }    // Verschiebekonstruktor
  X& operator=(X x)                  // Zuweisungsoperator
  { std::swap(p, x.p);               // copy (or move) & swap
    return *this;
  }                                  // Kopie wird freigegeben
  ~X() { delete p; }
private:
  T *p = nullptr;
};
```

[43] Resource Acquisition is Initialization (*RAII*), *Rule of the Big Three* (or Five)

**Inkrement- und Dekrementoperatoren** können als vor- und nachgestellte Operatoren definiert werden. Durch ein nachgestelltes, nichtbenutztes `int`-Argument wird der Operator als Postfixvariante deklariert:

```
class Date {
public:
  Date& operator++();        // Prefix-            ++date;
  Date  operator++(int);     // Postfix-Inkrement: date++;
};
Date& operator--(Date&);     // oder auch global:  --date;
Date  operator--(Date&,int); //                    date--;
```

**Feldindexklammern** `[]` erlauben den Zugriff auf Datenelemente einer Klasse. Lese- (`const`-) und Schreibmethode können unterschiedlich sein:

```
class SafeArray {
public:
  // ...
  const T& operator[](int index) const { return data[index]; }
  T& operator[](int index) { return valid(index)? data[index]: nowhere; }
  // ...
private:
  T *data;
  int valid(int idx); // Test Indexgrenzen
  static T nowhere;   // falsche Schreibzugriffe gehen hierhin
};

void demo(T x, SafeArray a)
{
  x = a[10];  // Lesezugriff:    operator[]() const
  a[10] = x;  // Schreibzugriff: operator[]()
}
```

**Funktionsklammern** `()` lassen Objekte als *Funktoren* wie Funktionen agieren:

```
class Linear {
public:
  Linear(float slope=0, float offset=0) : m{slope}, n{offset} {}
  float operator()(float x) { return m*x + n; }
private:
  float m,n;
};

Linear flach{0.5, 2}; // Funktionsdefinition: y = 0.5*x + 2;
float y=flach(3);     // Aufruf Funktionsoperator ()
```

**Typkonverter** `operator` ⟨*Typ*⟩`();`
erlauben, Instanzen in einen anderen Typ umzuwandeln. Wie bei einparametrigen Konstruktoren kann gefordert werden, dass die Umwandlung ausdrücklich angefordert wird:

```
struct Bruch {
  long z,n;
  explicit operator double() const { return double(z)/n; }
};

Bruch b{21,4};
double d{b}; // oder: auto d=double(b); ==> 5.25
```

**Literalsuffixe** wandeln eine Literalkonstante (Zeichenkette, Zahl) in einen anderen Typ um, z.B. `"Hallo"s` (wird ein `std::string`) oder `3.0+2.0i` (komplexe Zahl).[44] Sie werden als globale Funktionen definiert:

```
std::string operator""s (const char* s, std::size len)
{ return std::string(s, len); }         // in namespace std::literal
```

**Regeln und Hinweise für die Operatordefinition**

- Operatoren für eingebaute Typen sind nicht änderbar.
- Es lassen sich keine neuen Operatorzeichen definieren. Rangfolge und Anzahl der Operanden sind nicht veränderbar.
- Die Operatoren `. .* :: ?:` lassen sich nicht überladen.
- Die Operatoren `= [] () ->` sind nur als Methoden überladbar. Alle anderen Operatoren lassen sich als Methode oder global (`friend`) überladen.
- Operatoren wie `+=`, die Zugriff auf die private Daten eines Linkswertes (*lvalue*) benötigen, lassen sich meist besser als Methode implementieren, binäre Operatoren mit gleichartigen linkem und rechten Operanden dagegen global.
- Alle Operatormethoden außer `=` werden vererbt. ▶C.9.4
- Der Funktionsoperator `()` kann mit unterschiedlichen Parametern und beliebiger Parameterzahl überladen werden.
- Typkonverter sind parameterlose Methoden mit implizitem Rückgabetyp. Typkonversionen und Konstruktoren sollten keine Zirkel bilden, sonst beschwert sich der Compiler über Zweideutigkeiten.
- Überladene Operatoren sollten ihrer gewohnten Bedeutung nahekommen, z.B. sollten Vergleiche Wahrheitswerte liefern, der Kommaoperator den rechten Operanden. Der Zeigeroperator `->` sollte einen Zeiger auf eine Struktur oder Klasse liefern.

[44] Nutzerdefinierte Suffixe müssen mit einem Unterstrich beginnen.

### C.9.4 Vererbung

*Es ist niemand gestorben.*
– Niklaus Wirth

**Erweiterungen** von Klassendefinitionen dienen zwei unterschiedlichen Zielen. Klassen und Strukturen können ihre Attribute und Methoden an *abgeleitete* Klassen *vererben*[45],

1. um gleichartiges oder ähnliches Verhalten verschiedener, aber verwandter Klassen auszudrücken (gemeinsame Schnittstelle und Konkretisierung / Spezialisierung von Methoden) und so die Schnittstellenkomplexität zu reduzieren oder
2. um einmal geschriebene Methoden und definierte Datenfelder in verschiedenen Klassen zu nutzen (gemeinsame Codenutzung und Erweiterung).

Eine abgeleitete Klasse kann *Erbe* mehrerer *Basisklassen* sein (*Mehrfachvererbung*).

**Syntax** `class` ⟨*abgeleitete Klasse*⟩ `:` ⟨*Basisklassenliste*⟩ `{` ⟨*Erweiterungen*⟩ `};`

**Basisklassen** bilden den Ausgangspunkt der Vererbung.

```
class Fehler {
public:
  Fehler(std::string datei, int zeile) : datei{datei}, zeile{zeile} {}
  void warum(std::ostream& os) const
  {
    os << "Fehler in " << datei << ':' << zeile << endl;
  }
private:
  std::string datei;
  int zeile;
};
```

Diese Klasse könnte genutzt werden,

```
void report(Fehler& f)
{
  f.warum(std::cerr);
  std::cerr << "Programm gestoppt.\n";
  exit(1); // fascist solution
}
```

um Programmfehler mit Meldungen wie dieser zu lokalisieren:

```
Fehler in XYZ.CPP:nnnn
Programm gestoppt.
```

[45] In anderen Sprachen wird Vererbung Ableitung, Spezialisierung, Typerweiterung oder Typkoerzion genannt.

**Abgeleitete Klassen** können Methoden ändern oder ergänzen und / oder Datenfelder hinzufügen:

```
class NullDivision : public Fehler {
public:
  Nulldivision(std::string datei, int zeile) : Fehler{datei, zeile} {}
  void warum(std::ostream& os) const
  {
    Fehler::warum(os);
    os << "Division durch Null" << endl;
  }
};
```

Die Übergabe von Werten an Basiskonstruktoren erfolgt bei Bedarf nach dem Doppelpunkt : im Konstruktor. Zuerst werden die Basiskonstruktoren in der Reihenfolge ihrer Deklaration aufgerufen, dann neue Elemente initialisiert, zum Schluss der Konstruktorrumpf ausgeführt. Destruktoren verfahren in umgekehrter Reihenfolge (ohne expliziten Aufruf).

Ererbte Methoden lassen sich auch nach dem Überschreiben mit qualifiziertem Bezeichner ⟨*Basisklassenname*⟩`::`⟨*Methode*⟩`()` ansprechen.

Die abgeleitete Klasse erzeugt in

```
void demo()
{
  NullDivision div0{__FILE__,__LINE__};
  div0.warum(std::cerr);
}
```

folgende Standardfehlerausgabe:

```
Fehler in XYZ.CPP:nnnn
Division durch Null
```

**Öffentliche Vererbung** drückt dagegen Verwandtschaft zwischen Klassen aus und bedeutet, dass eine Instanz der abgeleiteten Klasse überall dort eingesetzt werden kann, wo eine Instanz der Basisklasse als Referenz oder über Zeiger erwartet wird:[46]

„Die `NullDivision` ist ein `Fehler`."

```
class NullDivision : public Fehler { /* ... */ };

float dezimalbruch(long z=0, long n=1)
{
  if (n == 0) report(NullDivision{__FILE__,__LINE__});
  return float(z)/n;
}
```

[46] Eine abgeleitete Instanz sollte nicht als Basisklassenwert zugewiesen werden. Die Daten der Instanz werden auf die Größe der Basisklasseninstanz zurechtgestümmelt (*object slicing*)!

Tabelle 9: Zugriffskontrolle bei Vererbung.

| Methode war in Basisklasse | Vererbungsart war `private` | `protected` | `public` |
|---|---|---|---|
| | Methode ist in abgeleiteter Klasse | | |
| `private` | nicht verwendbar | nicht verwendbar | nicht verwendbar |
| `protected` | nicht verwendbar | `protected` | `protected` |
| `public` | `private` | `protected` | `public` |

**Nicht-öffentliche Vererbung** eignet sich zur schrittweisen Implementierung:

„`NullDivision` wurde implementiert mit Hilfe von `Fehler`.“

Klassen erben privat, Strukturen öffentlich, falls nicht anders in der Basisklassenliste angegeben:

```
class NullDivision : private Fehler { /* ... */ };
```

**Die Zugriffskontrolle** regelt bei der Vererbung, ob und wie die Erben Zugriff auf die Methoden der Basisklassen haben (Tabelle 9). Eine nachfolgende Heraufstufung auf die bisherige Zugriffsstufe

```
class Base {
public:
  float z;
  void publ();
protected:
  int y;
  void prot();
private:
  char x;
  void priv();
};
```

ist bei nicht-öffentlicher Vererbung möglich durch die Angabe des qualifizierten Namens, jedoch nicht höher:

```
class Derived : private Base {
public:
  Base::z;
  Base::publ;
protected:
  Base::y;
  Base::prot;
};
```

Willkürliche Herabstufung ist nicht erlaubt.

**Virtuelle Methoden** garantieren, dass die verwandten Objekte mit überschriebenen Methoden tatsächlich (engl. virtual) richtig funktionieren.

Die Funktion `dezimalbruch()` sollte bei Aufruf mit `n=0` ausgeben:

```
Fehler in XYZ.CPP:nnnn
Division durch Null
Programm gestoppt.
```

Beim Zugriff auf Instanzen über Basisklassenzeiger oder -referenzen wird die zur aktuellen Instanz (*Nachkomme*) gehörende Methode nur dann aufgerufen, wenn sie in der Basisklasse als `virtual` deklariert wurde.

```
class Fehler {
  // ...
  virtual void warum(std::ostream& os) const;
  virtual ~Fehler() {}
};
```

Sonst bleibt es bei der Meldung der Basisklasse. Aus genau demselben Grund sollte jede Basisklasse mit virtuellen Methoden auch einen *virtuellen Destruktor* erhalten — um eine übergebene Instanz korrekt abbauen zu können und nicht nur die Basisinstanz zu vernichten (Gefahr eines Ressourcenlecks). Es ist erlaubt, virtuelle Methoden als überschreibend (`override`) zu einer virtuellen Basismethode mit gleicher Signatur zu kennzeichnen. Es ist ein Fehler, als abschließend (`final`) markierte virtuelle Methoden zu überschreiben:

```
class NullDivision : public Fehler {
  // ...
  void warum(ostream& os) const override final; // allerletzte Fassung
};
```

**Abstrakte Methoden** sind virtuelle Methoden, für die in der Basisklassendeklaration noch keine „vernünftige“ Implementierung (`= 0`) angegeben werden kann:

```
class Process {
public:
  Process();
  virtual ~Process();
  void start();
  void stop();
  virtual void run() = 0;
  void wait(long milliseconds);
  // ...
};
```

Klassen mit abstrakten Methoden (*abstrakte Basisklassen*) definieren einheitliche Schnittstellen und bilden damit *Wurzeln* in *Klassenhierarchien*. Instanzen mit dieser Schnittstelle können nur von abgeleiteten Klassen gebildet werden, die *alle* abstrakten Methoden der Basisklassen mit einer Neudefinition überschreiben.

**Mehrfachvererbung** erlaubt, einen Klassenbegriff mehreren Basiskategorien zuzuordnen: Ein Dateifehler im Netz kann sowohl als Dateifehler als auch als Netzwerkfehler behandelt werden. Datenströme, die sowohl Ein- und Ausgabe beherrschen, werden durch Einmischen (engl. *mix-in*) aus Ein- und Ausgabeströmen erzeugt:[47]

```
class NetFileError : public NetworkError, public FileError { /* ... */ };
class iostream : public istream, public ostream { /* ... */ };
```

Mehrfachvererbung bereitet Probleme[48],

- weil die relativen Adressen der Datenfelder und virtuellen Methodentabellen (der Offset bei Up-, Down- und Sidecasts) von der Deklarationsreihenfolge der Basisklassen abhängen.

  Wenn nötig, sollte dafür `dynamic_cast<Derived*>(base_ptr)` genutzt werden.

- wenn Basisklassen gleichnamige Komponenten haben. Dann müssen diese mit ihrem Klassennamen qualifiziert werden.

```
void namenskonflikt1()
{
  struct Base1 { int x; };
  struct Base2 { int x; };
  struct Derived : Base1, Base2 {} d;

  // d.x = 0; ==> Fehler!
  d.Base1::x = 1;
  d.Base2::x = 2;
}
```

- wenn Basisklassen gleichnamige Funktionen mit gleichartigen Parametern haben.[49] Diese können überschrieben werden, um zu entscheiden, welche gemeint ist.

```
void namenskonflikt2()
{
  struct Base1 { void f() {} };
  struct Base2 { void f() {} };
  struct Derived : Base1, Base2 {
    void f() { Base1::f(); }
  } d;

  d.f(); // ah!
}
```

[47] Wenigstens im Prinzip. In <iostream> sieht das etwas komplexer aus.
[48] Einige Sprachdesigner würden Mehrfachvererbung deshalb gern verbieten.
[49] Sonst sind sie anhand der Parameter unterscheidbar.

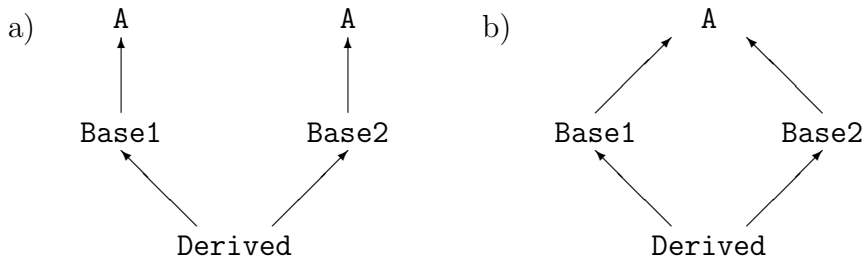



Abbildung 3: Mehrfachvererbung (a) ohne und (b) mit virtueller Basisklasse `A`.

- wenn Basisklassen von einem gemeinsamen Vorfahren abstammen (Abb. 3a). Dieser Vorfahr ist dann mehrfach in der abgeleiteten Klasse vorhanden. Das kann „doppelte Buchführung“ bedeuten.

```
void namenskonflikt3()
{
  struct A { int x; };
  struct Base1 : A { };
  struct Base2 : A { };
  struct Derived : Base1, Base2 { } d;

  // d.x = 0; ==> Fehler!
  d.Base1::x = 1;
  d.Base2::x = 2;
}
```

**Virtuelle Basisklassen** erzeugen nur eine Instanz, auch von sie über mehrere Basisklassen vererbt werden (Abb. 3b).

```
void virtuelle_Basis()
{
  struct A { int x; };
  struct Base1 : virtual A { };
  struct Base2 : virtual A { };
  struct Derived : Base1, Base2 { } d;

  d.x = 0; // ah!
}
```

Falls es keinen Standardkonstruktor für die virtuelle Basisklasse gibt, muss ein Konstruktor für `A` explizit und vor den Konstruktoren von `Base1` und `Base2` aufgerufen werden.

## C.10 Ausnahmebehandlung

### C.10.1 Ausnahmen werfen

`throw` ⟨*Ausnahme*⟩;
Eine *Ausnahme* (engl. exception) ist eine Situation, in der das Programm nicht wie erwartet fortgesetzt werden kann. Gleichzeitig besteht an dieser Stelle kaum die Chance, geeigneter auf diese Situation zu reagieren als durch das *Werfen* (engl. throw) einer ⟨*Ausnahme*⟩, mit der der Programmlauf unterbrochen wird. ⟨*Ausnahme*⟩ ist ein Objekt beliebigen Typs, das Informationen über die Ausnahmesituation enthalten kann.

```
float dezimalbruch(long z=0, long n=1)
{ if (n == 0) throw NullDivision{__FILE__,__LINE__};
  return float(z)/n;
}
```

### C.10.2 Ausnahmen oder Ausnahmefreiheit erklären

In Funktionen können Ausnahmen aller Art geworfen werden. Funktionen können sich jedoch verpflichten, keine oder nur bestimmte Ausnahmen zu werfen:[50]

```
float mul(float a, float b) noexcept { return a*b; }
float add(float a, float b) throw()  { return a+b; } // geächtet
```

Wird dennoch eine Ausnahme geworfen, führt `std::unexpected()` zum Programmende.

### C.10.3 Ausnahmen abfangen

Nach dem Werfen einer Ausnahme werden die bis zu diesem Programmpunkt auf dem *Stapel* entstandenen Variableninstanzen abgebaut (*stack unwinding*) und das Programm so kontrolliert beendet. (Indirekte) Aufrufer von Ausnahmen werfenden Funktionen können dem unmittelbaren Programmabbruch durch Einschließen der unsicheren Aufrufe in einen `try`-Block vorbeugen. Diesem Block folgen unmittelbar ein oder mehrere `catch`-Blöcke, die für die Reaktion auf Ausnahmen bestimmter Klassen zuständig sind. Der Typ der geworfenen Ausnahme wird mit den `catch`-Parametern verglichen. Beim ersten passenden Typ einer `catch`-Bedingung wird die Ausnahme *gefangen* (engl. catch), der Block abgearbeitet und dann nach dem letzten `catch`-Block normal fortgesetzt. Trifft keine Bedingung zu, geht der Ausnahmezustand weiter.

`try{` ⟨*unsichere Anweisungen*⟩ `}`
`catch(` ⟨*Typ*⟩ ⟨*name*⟩$_{opt}$ `){` ⟨*Anweisungen*⟩ `}`
⟨*weitere catch-Blöcke*⟩

[50] Nichtleere throw-Deklarationen haben sich als problematisch erwiesen [Herb Sutter: Guru of the Week, http://www.gotw.ca/publications/mill22.htm (2002)]. Wegen fehlender Meta-Information sind Compiler nicht in der Lage, die Deklarationen bei der Übersetzung durchzusetzen. In C++17 wurden nichtleere `throw`-Deklarationen entfernt. Mit `noexcept` wird versucht, diesen Design-Fehler auszubügeln.

```
void ausnahmebehandlung()
{
  try
  { // unsicherer Block:  u.a. Aufruf von dezimalbruch()
    // ...
  }
  catch(NullDivision& f)
  { cerr << f.warum() << endl;
  }
  catch(Fehler& f)
  { cerr << f.warum() << endl;
  }
  catch(HeisseKartoffel&)
  { cerr << "Autsch!" << endl;
    throw;     // weiterwerfen
  }
  catch(...)   // Ausnahmen beliebigen Typs
  { cerr << "Noch ein anderer Fehler!" << endl;
  }
  // hier geht's ordentlich weiter
}
```

Eine gefangene Ausnahme kann mit `throw;` weitergeworfen werden, falls die Situation in diesem Quelltextabschnitt nicht vollständig geklärt werden kann.[51] Konstruktoren können Ausnahmen in der Initialisiererliste abfangen:

```
class Zahl {
public:
  Zahl(long z=0, long n=1);
private:
  float wert;
};

Zahl::Zahl(long z, long n)
try
: wert{dezimalbruch(z,n)}
{
  // Konstruktorrumpf
}
catch(NullDivision&)
{
  wert = 0;
}
```

[51] Die Kombination aus `catch(...)` und `throw;`, in anderen Sprachen als `finally`-Block bezeichnet, wird in C++ kaum benötigt, da Destruktoren die Aufräumarbeiten erledigen (*RAII*-Prinzip).

## C.11 Freispeicherverwaltung

### C.11.1 Container und smarte Zeiger

Die Ablage von Objekten im *Freispeicher* ist vorteilhaft, wenn ihre Anzahl beim Schreiben des Programms nicht absehbar oder ihre Lebensdauer nicht an Funktionen gebunden ist. Der Speicher kann nach ihrem Abbau anderweitig genutzt werden. Container und smarte Zeiger regeln die Besitzverhältnisse an dynamischen Objekten und damit verbundene Aufgaben wie die Speicherfreigabe auch beim Auftreten von Ausnahmen.

►C.14

**Containerklassen** kapseln große Blöcke dynamischen Speichers.

```
#include <vector>

void gekapselte_Zeiger(long startwert)
{
  std::vector<long> v(50, startwert); // 50 Kopien von Startwert
  for (auto e : v) std::cout << v[i] << ' ';
  std::cout << '\n';
}                                     // automatische Speicherfreigabe
```

**Smarte Zeiger** sind Besitzer der Einzelobjekte, auf die sie zeigen und vernichten sie, wenn sie selbst ungültig werden. Die Funktionen `std::make_shared<T>(`⟨*Parameter*⟩`)` und `std::make_unique<T>(`⟨*Parameter*⟩`)` erzeugen dynamisch ein Objekt mit den angegebenen Konstruktor-Parametern und kapseln es in einem `std::shared_ptr<T>` bzw. einem verschiebbaren, aber nicht kopierbaren `std::unique_ptr<T>`:

```
#include <memory>

void sichere_einzelne_Zeiger()
{
  auto p = std::make_unique<int>(1);
  auto q = std::make_unique<int>(2);  // p = q; nicht erlaubt
  p = std::move(q);         // expliziter Besitzwechsel, q ist leer
  std::cout << *p << '\n'; // 2
}                           // automatische Freigabe

void sichere_gemeinsam_genutzte_Zeiger()
{
  auto p = std::make_shared<int>(1);
  auto q = std::make_shared<int>(2);
  p = q;    // p,q zeigen auf int(2), int(1) wird freigegeben
  *q = 3;
  std::cout << *p << '\n'; // 3
}                           // automatische Freigabe
```

**Zeiger in Klassen** können als smarte Zeiger geführt werden, wenn der referenzierte Speicher im Besitz der Instanz ist (*RAII-Prinzip*). Ein `std::unique_ptr<T>` als Attribut in Klassen verhindert Kopie und Zuweisung, sofern diese für die Klasse nicht definiert werden. Sie können ausdrücklich verboten werden:

```
struct Besitzer {                        // Objekte mit Verschiebesemantik
  Besitzer () : meine{std::make_unique<Teil>()} {}
  // Besitzer(const Besitzer &) = delete;
  // Besitzer operator=(const Besitzer &) = delete;
private:
  std::unique_ptr<Teil> meine;
};
```

Ein `std::shared_ptr<T>` erzeugt flache Kopien ohne weiteren Aufwand, aber im Gegensatz zu blanken Zeigern ist die Speicherfreigabe klar geregelt:

```
struct Nutzer {                         // Der Letzte macht das Licht aus.
  Nutzer () : gemeinsam{std::make_shared<Teil>()} {}
private:
  std::shared_ptr<Teil> gemeinsam;
};
```

**Polymorphe Container** sind durch Kombination von Containerklassen mit smarten Zeigern einfach zu realisieren; smarte Zeiger beachten dabei Vererbungsregeln:

```
class Animal {
public:
  virtual ~Animal() {};
};
class Sheep : public Animal {};
class Wolf  : public Animal {};

void polymorph()
{
  auto sheep = std::make_shared<Sheep>();
  auto wolf  = std::make_shared<Wolf>();
  auto zoo   = std::vector<std::shared_ptr<Animal>>{ sheep, wolf };
  auto flock = std::vector<std::shared_ptr<Sheep>>(10, sheep);
                                         // 10 Zeiger zum selben Schaf
  if (auto inSheepsClothing = std::dynamic_pointer_cast<Sheep>(wolf))
    flock.push_back(inSheepsClothing); // kommt nicht rein, da nullptr
}
```

### C.11.2 Hinter den Kulissen

**Die Operatoren `new` und `delete`** erschaffen und vernichten dynamisch Instanzen eines vorgegebenen Typs im Freispeicher.

⟨*Zeiger*⟩ `=` `new` ⟨*Typ*⟩(⟨*Parameter*⟩)$_{opt}$`;`
liefert einen Zeiger, über den auf die Instanz zugegriffen wird. Geeignete ⟨*Parameter*⟩ initialisieren die Instanz. Fehlen diese, wird der Standardkonstruktor aufgerufen. Grunddatentypen ohne Initialisierer enthalten undefinierte Werte.[52] Jede dynamische Instanz muss mit

`delete` ⟨*Zeiger*⟩`;`
irgendwann wieder abgebaut werden, um „geborgten“ Speicher zurückzugeben.

```
int *ip = new int(9);
std::cout << *ip  << '\n'; // Ausgabe: 9
delete ip;
```

Dynamische Felder mit ⟨*Anzahl*⟩ Elementen werden durch

⟨*Zeiger*⟩ `=` `new` ⟨*Typ*⟩`[`⟨*Anzahl*⟩`]`;
`delete []` ⟨*Zeiger*⟩`;`
mit dem Standardkonstruktor angelegt bzw. vernichtet.

```
int *ip = new int[9];
// Feld mit den Elementen ip[0] ... ip[8]
delete [] ip;
```

Mit `new[]` erzeugte Felder sind immer mit `delete[]` zu löschen, mit `new` erzeugte einzelne Instanzen immer mit `delete`. Vertauschungen bringen den Freispeicher durcheinander und führen zu Fehlfunktionen des Programms oder Systems.

**Speicherknappheit** führt zum Scheitern von Anforderungen:

```
T *q = new T[1000000];                // wirft bad_alloc-Ausnahme
T *p = new(std::nothrow) T[1000000]; // liefert 0
```

Die Ausnahme muss irgendwo gefangen werden, ist beim Schreiben von Bibliotheken aber bequemer, da der folgende Quelltext von gültigen Zeigern ausgehen kann. Bei der `nothrow`-Variante muss vor der Nutzung immer erst auf Gültigkeit `p!=0` geprüft werden.[53]

[52] Eingebaute Typen werden in C++ geringfügig schlechter unterstützt, da sie wegen der Forderung nach Effizienz und Kompatibilität zu C keinen Standardkonstruktor besitzen.

[53] Das Verhalten des `new`-Operators ist änderbar. Ist kein Speicher verfügbar, wird eine Funktion `void newhandler()` aufgerufen, die entweder weiteren Speicher bereitstellt und dann zu `new` zurückkehrt, eine `bad_alloc`-Ausnahme wirft bzw. `exit()` aufruft. Eigene Newhandler können mit `set_new_handler(myhandler)` installiert werden. Der bisherige Handler wird zurückgeliefert und kann aufbewahrt werden, um das vorherige Verhalten wiederherzustellen.

**Platzierungssyntax** ermöglicht, eine eigene Speicherverwaltung aufzubauen. Eine Instanz kann an einem vorgegebenen Speicherplatz erstellt werden.

```
void sie_werden_platziert()
{
  char buffer[sizeof(T)];
  T* p = new(buffer) T(startwert); // Konstruktoraufruf
  // ... *p verwenden
  p->~T();                         // Destruktoraufruf, nicht delete!
}
```

**Fehler mit dynamischen Zeigern** sind aufgrund ihrer verheerenden Wirkungen gefürchtet und leider auch schwer zu erkennen:[54]

1. Speicherleichen sind nicht mehr erreichbar und können auch nicht mehr vernichtet werden (Ressourcenleck).
2. Durch Löschen des einen Zeigers werden weitere Zeiger auf dieselbe Instanz ungültig (baumelnde Zeiger).
3. Wildes Schreiben in anderweitig genutzten Speicher kann weitere Daten durcheinanderbringen (undefiniertes Verhalten).
4. Schreiben in Speicher, der dem Programm nicht mehr gehört, führt zu undefiniertem Verhalten, bestenfalls Absturz („Schutzverletzung“ oder „segmentation fault“).
5. Doppelte Freigaben korrumpieren die Speicherverwaltung und können im Systemabsturz enden.

```
void zeigerkatastrophe(T x)
{
  T *p = new T;
  T *q = new T;

  p = q;          // Fehler 1: altes *p ist Speicherleiche
  delete q;       // Fehler 2: p zeigt auf gekilltes Objekt
  *p = x;         // Fehler 3: wildes Schreiben in Freispeicher
  delete p;       // Fehler 4: doppelte Freigabe, Absturz ?
}
```

Vor allem die nicht-lokale Wirkung von Zeigerfehlern gestaltet die Fehlersuche schwierig: Die Ursache kann irgendwo im Programm liegen. *Post-mortem-Analyse* erfordert tiefgründige Kenntnisse und viel Zeit zum Debuggen. Überwachungswerkzeuge wie *Valgrind* können helfen, solche Fehler aufzuspüren.

[54] Niemand erzeugt solche Fehler mutwillig. Doch schon die vergessene Neudefinition von Kopierkonstruktor und Zuweisungsoperator von Klassen mit dynamischen Zeigern führt zu diesem Verhalten.

## C.12 Zeichenketten

### C.12.1 C-Zeichenketten

```
char ptr1[] = { 'H', 'a', 'l', 'l', 'o', '\0'  };
const char *ptr2 = "Hallo";  // Zeichenkettenliteral nicht änderbar
```

*Nullterminierte Zeichenfolgen* werden mit einem Zeiger auf den Anfang des Feldes angesprochen. Sie müssen ausreichend groß sein und mit `'\0'` abgeschlossen bleiben (sonst Pufferüberlauf und undefiniertes Verhalten). Zeiger müssen bei den Operationen gültig sein (Gefahr baumelnder Zeiger, Prüfung auf Nullzeiger). Speicherbereitstellung und Freigabe liegen in der Verantwortung des Programmierers (Gefahr von Speicherlecks).

Als Funktionsparameter entarten sie zu Zeigern (Referenzverhalten), können nur als Zeiger zurückgegeben werden, müssen als Parameter durchgereicht oder dynamisch verwaltet werden. Die Rückgabe von Zeigern auf lokaler Zeichenfelder führt zu baumelnden Zeigern. Für *nur zu lesende* (Teil-)Zeichenbereiche bietet `<string_view>` eine Kapsel aus einem *nicht besitzenden* Zeiger auf den Anfang und der Anzahl folgender Zeichen an. Vergleiche und Suchmethoden wie bei `std::string` sowie Einkürzungen sind möglich. Hauptsächlich gedacht ist diese Kapsel für die Nutzung als Funktionsparameter[55]:

```
bool sv_test(std::string_view s) { return s == "all"; }

using std::literal;
sv_test("all");       // Zeichenkettenliteral
sv_test("all"s);      // std::string
sv_test("all"sv);     // string_view
sv_test({ptr2+1, 3}); // ptr, anzahl
sv_test("Hallo, Welt!"sv .remove_prefix(1).remove_suffix(8));
```

### C.12.2 Die Klasse `string`

C++ kapselt diese fehlerträchtigen Routineaufgaben in Klassen. Durch optimierende Compiler, Kopier- und Verschiebesemantik sind `std::string`-Objekte[56] auch als Wertparameter und Rückgabewerte tauglich; effektiv werden intern nur Zeiger weitergereicht. Die Klassenschnittstelle im Header `<string>` ist umfangreich. Auf den nächsten Seiten stehen der Übersichtlichkeit wegen folgende Parameternamen für die Typen:

```
std::string             s;
const std::string&      str;
const char*             ptr;
char                    c;
std::string::size_type  pos, n, pos2, n2;
std::string::iterator   first, last, first2, last2;
```

[55] Zeiger bleiben gefährlich. Nicht als Rückgabetyp einsetzen!

[56] Für Zeichenketten aus `wchar_t`-Zeichen `L"Hallo"` gibt es `std::wstring`s.

**Konstruktoren** schaffen einen neuen `std::string` aus

| | |
|---|---|
| `std::string(str, pos=0, n=npos)` | `str` (ab `str[pos]` mit max. `n` Zeichen), |
| `std::string(ptr)` | einer C-Zeichenkette `ptr` (`!=nullptr`) |
| `std::string(ptr, n)` | (mit `n` Zeichen oder bis zur Ende-Null), |
| `std::string(n, c)` | `n` Zeichen `c`, |
| `std::string(first, last)` | einem *Iterator*-Bereich `[first...last[`. |

```
void string_demo()
{
  const char *ptr = "Hello, world of strings!";

  std::string s1;              // leer
  std::string s2 = ptr;
  std::string s3(ptr);         // Kopie von "Hello, world of strings!"
  std::string s4(s2, 7);       // "world of strings!"
  std::string s5(s2, 7, 5);    // "world"
  std::string s6(24, '*');     // "************************"
  std::string s7(ptr, ptr+5);  // "Hello"

  std::cout << s7 << ", " << s4 << '\n'
            << s6 << '\n';
}
```

**Zuweisungen** mit `=` oder der Methode `assign(...)` weisen einem `std::string`

| | |
|---|---|
| `assign(str, pos=0, n=npos)` | `str` (ab `str[pos]` mit höchstens `n` Zeichen), |
| `assign(ptr)` | eine gültige C-Zeichenkette |
| `assign(ptr, n)` | (mit maximal `n` Zeichen), |
| `assign(n, c)` | `n` Zeichen `c`, |
| `assign(first, last)` | einen Bereich `[first...last[` zu. |

```
void zuweisung_demo()
{
  std::string s;
  const char *ptr = "Zeichenketten";

  s = std::string(ptr, 7);
  s = ptr;
  s = 'C';
  s.assign(10,'.');
  s.assign(ptr, 6);
  s.assign(ptr, ptr+7);

  std::cout << s << '\n';
}
```

**Verknüpfungen** von Strings

| | |
|---|---|
| `operator+=(str2)` | Anhängen |
| `operator+ (str1,str2)` | Verketten |

können auch mit C-Zeichenketten oder einzelnen Zeichen erfolgen:

```
s += str;
s += ptr;
s += c;
s = str1 + str2;
s = str + ptr;
s = ptr + str;
s = ptr + std::string(ptr); // ptr + ptr ist nicht erlaubt
s = c + str;
s = str + c;
s = c + std::string(1, c);  // c + c erzeugt nur ein char
```

Die Methoden `append(...)` hängen an einen existierenden `std::string`

| | |
|---|---|
| `append(str, pos=0, n=npos)` | `str` (ab `str[pos]` mit höchstens `n` Zeichen), |
| `append(ptr)` | eine C-Zeichenkette |
| `append(ptr, n)` | (mit maximal `n` Zeichen), |
| `append(n, c)` | `n` Zeichen `c`, |
| `append(first, last)` | einen Bereich `[first...last[` an. |

```
void anhaengen_demo()
{
  std::string s = "...";
  const char *ptr = "ketten";

  s += "Zeichen";
  s = s + ' ' + ptr;

  s.append(" wie Kletten");
  s.append(10,'.');

  std::cout << s << '\n';
}
```

**Einfügungen** schieben *vor* die Position `pos` bzw. *vor* den Iterator `iter_pos`

| | |
|---|---|
| `insert(pos, str, pos2=0, n=npos)` | `str` (ab `str[pos2]`, max. `n` Zeichen), |
| `insert(pos, ptr)` | eine gültige C-Zeichenkette |
| `insert(pos, ptr, n)` | (mit `n` Zeichen), |
| `insert(pos, n, c)` | `n` Zeichen `c`, |
| `insert(iter_pos, c = '\0')` | ein Zeichen `c`, |
| `insert(iter_pos, n, c)` | `n` Zeichen `c`, |
| `insert(iter_pos, first, last)` | einen Bereich `[first...last[` ein. |

```
void einsetzen_demo()
{
  std::string s = "immer";
  char ptr[] = "=>> noch was <<=";
  s.insert(0, "Mann-oh-mann", 0, 3);
  s.insert(3, "kann ");
  s.insert(3, 1, ' ');
  s.insert(s.end(), ptr, ptr+strlen(ptr) );
  std::cout << s.insert(s.length(), "einfuegen.") << '\n';
}
```

**Löschmethoden** entfernen

| | |
|---|---|
| `erase(pos=0, n=npos);` | ab `s[pos]` maximal `n` Zeichen, |
| `erase(iter_pos);` | das Zeichen `*iter_pos`, |
| `erase(first, last);` | den Bereich `[first...last[`. |

```
void loeschen_demo()
{
  std::string s = "Alles Ueberfluessige wird unverzueglich geloescht.";
  s.erase(s.end()-1);
  s.erase(s.begin()+6, s.end()-9);
  s.erase(5, 2);
  std::cout << s << '\n';
}
```

**Ersetzen** lassen sich ab `s[pos]` `n` Zeichen oder der Bereich `[first...last[` durch

| | |
|---|---|
| `replace(pos, n, str, pos2=0, n2=npos);` | `str[pos2]...str[pos2+n2]`, |
| `replace(pos, n, ptr);` | eine C-Zeichenkette, |
| `replace(pos, n, ptr, n2);` | `ptr[0]...ptr[n2]`, |
| `replace(pos, n, n2, c);` | `n2` Zeichen `c`, |
| `replace(first, last, str);` | ⋮ |
| `replace(first, last, ptr);` | |
| `replace(first, last, ptr, n2);` | |
| `replace(first, last, n2, c)` | |
| `replace(first, last, first2, last2)` | den Bereich `[first2...last2[`. |

```
void ersetzen_demo()  // nach [B. Stroustrup: C++, 3rd edn]
{
  std::string s1 = "Das klappt nur dann, wenn du fest daran glaubst.";
  std::string s2 = "nicht";
  s1.replace(s1.find("nur dann"), 8, s2);     // wird kuerzer
  s1.replace(s1.find("fest"), 4, s2);         // wird laenger
  s1.replace(s1.end()-1, s1.end(), 10, '.' );
  std::cout << s1 << '\n';
}
```

**Suchmethoden** suchen ab `s[pos]` in den folgenden maximal `n` Zeichen

- einen Teilstring `str`, eine C-Zeichenkette `ptr` oder ein Zeichen `c`

| von vorn | von hinten |
|---|---|
| `find(str, pos = 0)` | `rfind(str, pos = npos)` |
| `find(ptr, pos, n)` | `rfind(ptr, pos, n)` |
| `find(ptr, pos = 0)` | `rfind(ptr, pos = npos)` |
| `find(c, pos = 0)` | `rfind(c, pos = npos)` |

- die

| erste Übereinstimmung | letzte Übereinstimmung |
|---|---|
| `find_first_of(str, pos = 0)` | `find_last_of(str, pos = npos)` |
| `find_first_of(ptr, pos, n)` | `find_last_of(ptr, pos, n)` |
| `find_first_of(ptr, pos = 0)` | `find_last_of(ptr, pos = npos)` |
| `find_first_of(c, pos = 0)` | `find_last_of(c, pos = npos)` |
| erste Nicht-Übereinstimmung | letzte Nicht-Übereinstimmung |
| `find_first_not_of(str, pos = 0)` | `find_last_not_of(str, pos = npos)` |
| `find_first_not_of(ptr, pos, n)` | `find_last_not_of(ptr, pos, n)` |
| `find_first_not_of(ptr, pos = 0)` | `find_last_not_of(ptr, pos = npos)` |
| `find_first_not_of(c, pos = 0)` | `find_last_not_of(c, pos = npos)` |

  mit einem Zeichen der Zeichenmenge `str`, `ptr` oder `c`.

Sie liefern die Anfangsposition der gefundenen Teilkette bzw. des Zeichens, auf das die geforderte Bedingung zutrifft; andernfalls wird `std::string::npos` zurückgegeben:

```
void suchen_demo()
{
  std::string s = "Entente cordial", NT = "nt"; // "Not There" ?
  const char vocals[] = "AEIOUaeiou";
  std::string::size_type pos = 0;

  pos = s.find (NT);                    // pos == 1
  pos = s.rfind(NT);                    // pos == 4
  pos = s.find_first_of(vocals);        // pos == 3
  pos = s.find_last_of (vocals);        // pos == 13
  pos = s.find_first_not_of(vocals);    // pos == 1
  pos = s.find_last_not_of (vocals);    // pos == 14

  if (pos != std::string::npos) // gefunden
  {
    std::cout
      << "Wir finden den letzten Konsonanten (markiert durch ^) in :\n"
      << s << '\n'
      << string(pos,' ') << '^' << '\n';
  }
}
```

**Zugriff auf ein einzelnes Zeichen** an der Stelle `pos` zum Schreiben oder Lesen erfolgt
- `s[pos]` ungeprüft oder
- `s.at(pos)` geprüft.

Wenn `pos` hinter dem Ende liegt, wirft `at()` eine `out_of_range`-Ausnahme.

**Iteratormethoden** liefern abhängig von Durchlaufrichtung und Konstantheit

| | | |
|---|---|---|
| `begin()`, `end()` | (vorwärts) | `std::string::iterator` oder `std::string::const_iterator`, |
| `rbegin()`, `rend()` | (rückwärts) | `std::string::reverse_iterator` oder `std::string::const_reverse_iterator` |

als *Random-Access-Iteratoren* auf den Anfang und *hinter* das Ende des `std::string`-Inhalts. Diese Iteratoren verhalten sich wie Zeiger in die Zeichenkette. Die vorwärts gerichteten Iteratoren sind auch über globale Funktionen `begin(s)` und `end(s)` erreichbar. ▶C.14.1

```
void zugriff_demo()
{
  std::string s = "zeichen sind einzeln ansprechbar!";

  s[0] = 'Z';
  s.at(s.length()-1) = '.'; // letztes Zeichen

  std::string::iterator i = s.begin();  // oder begin(s)
  std::string::reverse_iterator ri = s.rbegin();

  while (i != s.end())  // oder end(s)
  {
    std::cout << *i++ << ' ';
  }
  std::cout << "\nIteratoren wandern hin ... und her:\n";

  while (ri != s.rend())
  {
    std::cout << *ri++ << ' ';
  }
  std::cout << '\n';
}
```

**Vertauschungen** wechseln die Inhalte aus:

```
void tauschen_demo()
{
  std::string s1 = "abc", s2 = "def";

  s1.swap(s2);       // hin ... (effizienter)
  std::swap(s1, s2); // und zurueck
}
```

**Vergleiche** (`== != < <= >= >`) zweier `std::string`s untereinander als auch mit C-Zeichenketten sind möglich. Die `compare()`-Methoden vergleichen in `s` bzw. ab `s[pos]` maximal `n` Zeichen mit

| | |
|---|---|
| `compare(str, pos2, n2)` | `str[pos2]...str[pos2+n2]` |
| `compare(pos, n, str, pos2=0, n2=npos)` | (bzw. dem String `str`) |
| `compare(ptr, n2)` | der C-Zeichenkette `ptr` |
| `compare(pos, n, ptr, n2 = npos)` | (maximal `n2` Zeichen) |

und liefern

| | |
|---|---|
| einen negativen Wert, | wenn `s` lexikographisch vor dem Argument kommt, |
| einen positiven Wert, | wenn `s` danach kommt, |
| Null | bei Übereinstimmung. |

```
void vergleich_demo(std::string name)
{
  std::string str = "Asterix";
  const char ptr[] = "Obelix";

  if (name == str || ptr == name)
    std::cout << "Die spinnen, die Roemer!\n";
  if (name.compare(str) > 0 && name.compare(ptr) < 0)
    std::cout << name << " zwischen " << str << " und " << ptr << ".\n";
}
```

**Ein- und Ausgaben** erfolgen mit den Stromoperatoren `>>` und `<<`. Eingelesen wird nur bis zum nächsten Whitespace, führende Whitespaces werden verschluckt. Die Funktion `getline()` liest dagegen alle Zeichen bis zum Auftreten des Begrenzers `delim` ein. Der Begrenzer wird aus dem `std::istream` ausgelesen, jedoch nicht in `s` aufgenommen.

```
void io_demo(char delim = '\n')
{
  std::string s = "Ich habe Hunger. Gib mir einen Keks! :";
  std::cout << s;                    // Kruemelmonster
  std::getline(std::cin, s, delim); // auch ohne delim = '\n'
  std::cout << s << '\n';
}
```

**Größe und Kapazität** eines `std::string` sind erfragbar und änderbar:

| | |
|---|---|
| `empty()` | liefert `true` bei Leerstring, |
| `size()` | Anzahl von Zeichen, |
| `length()` | dto., |
| `max_size()` | größte mögliche Zeichenzahl, |
| `capacity()` | Fassungsvermögen ohne Reallokation, |
| `reserve(n = 0)` | reserviert Speicher fuer `n` Zeichen, |
| `resize(n, c = '\0')` | kürzt oder füllt mit `c` auf, |
| `clear()` | leert den String. |

**Teil- und C-Zeichenketten** lassen sich aus einem `std::string` herausziehen:

| | |
|---|---|
| `substr(pos=0, n=npos)` | eine Teilkette als `std::string`, |
| `c_str()` | eine temporär gültige C-Zeichenkette (Zeiger), |
| `copy(ptr, n, pos = 0)` | kopiert max. `n` Zeichen ab `s[pos]` nach `ptr`. |

Die Umwandlung in eine C-Zeichenkette sollte nur erfolgen, wenn dies unumgänglich ist. Der von `c_str()` gelieferte Zeiger ist nur gültig, solange der liefernde `std::string` existiert und noch nicht geändert wurde. Die Methode wurde hauptsächlich eingeführt, weil viele (alte) Bibliotheksfunktionen einen `const char*` erwarten:

```
std::remove(oldFilename.c_str());          // #include <cstdio>
std::rename(oldFilename.c_str(), newFilename.c_str());
```

Bei `copy()` muss das Feld `ptr[]` genügend lang sein; eine abschließende Null wird nicht kopiert, diese muss der Aufrufer selbst anfügen.

```
void teilketten_demo(std::string s = "Apollo 13")
{
  char ptr[80];                              // gross genug ?
  std::string::size_t n = s.copy(ptr, 79); // anz kopierte Zeichen
  ptr[n] = '\0';                             // End-Null nicht vergessen!
  std::cout << ptr << '\n';
  std::cout << s.substr(0,6) << '\n';
}
```

**Konversion von Zahlen** in Zeichenketten erfolgt mit `std::to_string(zahl)`. Für die umgekehrte Richtung sind die Funktionen `std::stoX(s, ...)` vorgesehen. Der optionale zweite Parameter liefert die Position hinter dem letzten Zeichen der Zahl, der dritte Parameter bei Ganzzahlkonvertierungen legt die Zahlbasis fest.

```
void konvertieren(int i, long long ll, double d)
{
  int base = 16;
  size_t end;
  i  = std::stoi("0xff", &end, base);     // auch: stol(long, ...)
  ll = std::stoll("0xff", &end, base);    //       stoul(long long, ...)
  d  = std::stod("1.23", &end);           //       stof(float, ...)

  std::cout << std::to_string(i) << ' '   // noch nicht in gcc 4.7, VC10
            << std::to_string(ll) << ' '
            << std::to_string(f) << ' '
            << std::to_string(d) << '\n';
}
```

### C.12.3 Lokalisierung und Zeichenarten

*Internationalisierung* (Anpassung an bestimmte Besonderheiten von Sprachen, Regionen und Zeichensätzen) erlaubt, Ein- und Ausgaben landesspezifisch zu formatieren:[57]

```
#include <locale>
// ...
  std::locale loc("German"); // VC10
  std::cout.imbue(loc);
  std::cout << 1234.56;      // 1.234,56
```

Zeichenketten sprachabhängig korrekt sortieren, z.B. `Hase Häsin Hund Hündin`:

```
  std::string tiere[size];
  std::sort(tiere, tiere+size, loc);
```

auch Zeicheneigenschaften (`isspace`, `isprint`, `iscntrl`, `isdigit`, `isxdigit`, `isalpha`, `isupper`, `islower`, `isgraph`, `isblank`) hängen von der gewählten Region ab. Eigenheiten wie Sortierung, Zeichenart, Kodierung, Geld-, Zahl- und Zeitformate werden in Facetten[58] implementiert. Die Schnittstelle ist durch eigene Facetten erweiterbar:

```
class Umlaut : public std::locale::facet
{
public:
  static std::locale::id id;
  bool is_umlaut(char c) const { /* ... */ }
protected:
  ~Umlaut() {};
};

std::locale loc2(loc, new Umlaut); // neue Facette in Kopie einbauen

void zeichentyp(char c, const std::locale& loc)
{
  if (std::isalpha(c, loc)) std::cout << c << "ist Buchstabe." << '\n';
  if (std::has_facet<Umlaut>(loc) &&
      std::use_facet<Umlaut>(loc).is_umlaut(c))
    std::cout << c << "ist Umlaut!" << '\n';
}
```

[57] Die Namen der Sprachen und Regionen sind implementierungsabhängig: g++ 4.7 unterstützt unter Windows nur die Namen "`C`" und "`POSIX`". Unter Linux sind Locale-Namen wie "`de_DE.UTF-8`" gängig. Microsoft benutzt nicht standardisierte Bezeichner. `Boost.Locale` versucht, die Mängel der seit 1998 vorhandenen Bibliothek zu beseitigen.

[58] Rogue Wave (http://h30097.www3.hp.com/cplus/intzln.pdf) gibt dazu eine gute Einführung.

### C.12.4 Reguläre Ausdrücke

**Suchmuster** (reguläre Ausdrücke) ermöglichen effizientes Erkennen, Prüfen, Suchen, Filtern von und Ersetzen in Zeichenketten. In der Klasse `std::regex` wird ein endlicher Automat zur Prüfung regulärer Ausdrücke bereitgestellt. Fehlerhafte Ausdrücke werfen eine Ausnahme vom Typ `std::regex_error`. Eine Suche ist schon erfolgreich, wenn ein *Teil der Zeichenkette* mit dem regulären Ausdruck übereinstimmt:

```
#include <fstream>
#include <iostream>
#include <regex>
#include <string>

int main(int argc, char* argv[]) // Minimalversion von grep
{
  if (argc < 3) return 1;
  try
  {
    std::regex    re{argv[1]};
    std::ifstream in{argv[2]};
    std::string   line;

    while (std::getline(in, line))
    {
      if (std::regex_search(line, re)) std::cout << line << '\n';
    }
  }
  catch (std::regex_error& e)
  {
    std::cerr << '\"' << argv[1] << "\" : " << e.what() << '\n';
  }
}
```

**Validieren** prüft dagegen die Übereinstimmung („*match*“) der *gesamten Zeichenkette* mit den im Ausdruck festgelegten Regeln:

```
std::string s = "user@example.com";
std::regex rex;
rex = "[a-z0-9_-]+(\\.[a-z0-9_-]+)*@[a-z0-9_-]+(\\.[a-z0-9_-]+)+";

if (std::regex_match(s, rex)) std::cout << "eMail ok";
```

In regulären Ausdrücken auftretende Backslashes (siehe Tab. 10) müssen im Quelltext als `\\` entwertet werden. Sauberer ist dann die Darstellung als *raw string*:

```
rex = R"_([a-z0-9_-]+(\.[a-z0-9_-]+)*@[a-z0-9_-]+(\.[a-z0-9_-]+)+)_";
```

Tabelle 10: Ausgewählte Symbole in regulären Ausdrücken.

| | | | |
|---|---|---|---|
| `c` | normales Zeichen | `\|` | logisches Oder von Ausdrücken |
| `^` | Zeilenanfang, Negation in `[]` | `()` | gruppiert Teilausdrücke |
| `$` | Zeilenende | `[]` | definiert Klasse von Zeichen |
| `.` | jedes Zeichen außer `\n` | `*` | Ausdruck davor beliebig oft oder nie |
| `\d` | Ziffer `[0-9]` | `+` | Ausdruck mindestens 1 mal |
| `\D` | keine Ziffer `[^0-9]` | `?` | Ausdruck 0 oder 1 mal |
| `\w` | _ oder Buchstabe `[A-Za-z_]` | `{`$n$`}` | Ausdruck genau $n$ mal |
| `\W` | kein Buchstabe oder _ | `{`$n$`,}` | Ausdruck mindestens $n$ mal |
| `\s` | Whitespace `[\t\n\r\f]` | `{`$n$`,`$m$`}` | mindestens $n$, höchstens $m$ mal |
| `\S` | kein Whitespace | `\`$c$ | Zeichen $c$ ohne Sonderbedeutung: |
| `\xFF` | hexkodiertes Zeichen | | `^ . $ \| () [] * + ? \ /` |

**Ersetzen** liefert eine neue Zeichenkette, in der alle Fundstellen des regulären Ausdrucks durch die Ersatzzeichenkette ausgetauscht sind:

```
std::string original = "Drei Chinesen", ersatz = "a";
std::regex rex{"[aeiou]"};
std::cout << std::regex_replace(original, rex, ersatz) << '\n';
```

Mit dem optionalen Parameterwert `std::regex_constants::format_first_only` wird der Austausch auf die erste Fundstelle beschränkt.

**Übereinstimmende Bereiche** (*matches*) der Durchmusterung mit `regex_match` und `regex_search` können in einem `std::smatch`-Referenzparameter als Paare von Iteratoren auf das erste und hinter das letzte Zeichen hinterlegt werden.[59]

```
std::string s = "24.12.2011";
std::regex rex{R"_((\d{1,2})\.(\d{1,2})\.(\d{4}|\d{2}))_"};
std::smatch matches;
if (std::regex_search(s, matches, rex))
{
  std::string day = matches[1], month = matches[2], year = matches[3];
}
```

Die gefundene Gesamtkette wird in `matches[0]` notiert, durch runde Klammern eingegrenzte Teilausdrücke liefern weitere Paare `matches[i]` mit `i < matches.size()`:

```
matches:   [----0----]
           24.12.2011
           -- -- ----
           [1][2][-3-]
```

[59] Diese Paare sind zu `std::string` konvertierbar. Für die Klassen der `<regex>`-Bibliothek existieren Varianten mit `wstring` und C-Zeichenketten (`wregex`, `wsmatch`, `cmatch`, `wcmatch`). Die Funktionen sind als Schablonen realisiert.

Reguläre Ausdrücke („wildcards on steroids“) können recht komplex sein. Zum Testen empfiehlt sich ein kleines Programm wie das folgende[60]:

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
  while (true)
  {
    std::string expr, s;
    std::cout << "Ausdruck:     ";
    std::getline(std::cin, expr);
    if (expr == "exit") break;
    std::cout << "Zeichenfolge: ";
    std::getline(std::cin, s);
    try
    {
      std::regex rex{expr};
      std::smatch matches;
      if (std::regex_match(s, matches, rex))
      {
        for (int i = 0; i < matches.size(); i++)
        {
          std::string match = matches[i];
          std::cout << "\tmatches[" << i << "] = " << match << '\n';
        }
      }
      else std::cout << "Der Ausdruck \"" << expr
                     << "\" trifft auf \"" << s << "\" nicht zu.\n";
    }
    catch (std::regex_error& e)
    {
      std::cout << expr << " ist kein regulaerer Ausdruck: \""
                << e.what() << "\"\n";
    }
  }
}
```

[60] nach: http://onlamp.com/pub/a/onlamp/2006/04/06/boostregex.html

## C.13 Ein- und Ausgabe

### C.13.1 Datenströme

C++ bietet eine einheitliche Schnittstelle für ein- und ausgehende Daten:

```
#include <iostream>

void caesar_chiffre(std::istream& in, std::ostream& out)
{
  char c;
  while (in.get(c)) out << char(c+3);
}
```

Dabei ist es gleichgültig, woher die Daten kommen (`std::istream`) oder wohin sie gehen (`std::ostream`) — Geräte (Konsole), Dateien oder Zeichenketten (`std::string`):

```
std::string buf="123+1.23/4.5-.1";
std::istringstream in{buf};     // string als Eingabestrom
std::ofstream out("datei.txt"); // Ausgabe-Dateistrom
```

Erreicht wird dies durch eine Hierarchie von Klassen-Schablonen für beliebige Eingabezeichen. In Tabelle 11 sind die Spezialisierungen für den `char`-Typ angegeben. Strömen für *Wide-Character-* (`wchar_t`) -Zeichen ist jeweils noch ein `w` vorangestellt.

**Standard-Ein- und -Ausgabekanäle** sind während der gesamten Programmdauer mit vier Stromvariablen verbunden:

| | | | |
|---|---|---|---|
| Eingabe-Konsole: Tastatur | `std::cin` | Fehlerausgabe: Bildschirm | `std::cerr` |
| Ausgabe-Konsole: Bildschirm | `std::cout` | Fehlerausgabe (gepuffert) | `std::clog` |

Die Ströme können beim Programmaufruf[61] auch in Dateien oder Pipes gelenkt werden:

```
program.exe < input.dat > output.dat 2> fehler.log
```

**Ausgabeströme** `std::ostream` nehmen Daten aller Typen entgegen, für die ein Ausgabeoperator `operator<<()` definiert ist und schreiben die entsprechende Zeichenfolge.

```
void ausgabe(std::ostream& out, double d)
{
  out << d << " Hallo"; // Ausgabe
  out.put('\n');        // einzelnes Zeichen
  out.flush();          // Puffer leeren
  out.write(reinterpret_cast<char*>(&d), sizeof(d)); // Speicherinhalt
}
```

Die Methode `write(char*, size_t)` ist für die unformatierte (Low-Level-)Ausgabe von Binärdateien geeignet. Sie schreibt den Speicherinhalt byteweise und versucht nicht, Zeichenketten daraus zu machen. Da die Speicheranordnung der Grundtypen hardwarespezifisch ist, ist das Ergebnis jedoch nicht portabel.

[61] Beispiel aus UNIX (betriebssystemabhängig).

Tabelle 11: Im Namensraum `std` definierte Stromklassen zur Ein- und Ausgabe.

| | Eingabe | Ausgabe | beides |
|---|---|---|---|
| allgemein | `istream` | `ostream` | `iostream` |
| Dateien | `ifstream` | `ofstream` | `fstream` |
| Zeichenketten | `istringstream` | `ostringstream` | `stringstream` |

**Eingabeströme** `std::istream` verwandeln eingegebene (zulässige) Zeichenfolgen in Variablenwerte aller Typen, für die `operator>>()` definiert ist. Führende Whitespaces werden bei der formatierten Eingabe mit `>>` verschluckt (Voreinstellung).

```
void eingabe(std::istream& in, std::string& s, char& c, double& d)
{
  in >> d >> s;          // Eingabe
  std::getline(in, s);   // ganze Zeile (Zeilenumbruch wird verschluckt)
  in.get(c);             // einzelnes Zeichen
  in.putback(c);         // gelesenes Zeichen wieder in Puffer schieben
  c = in.peek();         // nachschauen, aber nicht auslesen
  in.read(reinterpret_cast<char*>(&d), sizeof(d)); // Bytes in Speicher
}
```

Unformatierte Eingaben aus Binärdateien erfolgen mit `read(char*, size_t)`. Die Eingabe von Zeichenketten stoppt beim nächsten Whitespace-Zeichen. Daher wird vom Einlesen in Zeichenfelder dringend abgeraten (Gefahr des Pufferüberlaufs):

```
char str[10]; // lang genug ?
in >> str;    // kann und wird schiefgehen
```

Die Methoden `getline()` und `get()` lesen auch führende Whitespaces in die Zeichenkette ein, begrenzen die Eingabelänge auf `n-1` Zeichen und hängen stets eine Null an. Werden keine oder `n-1` Zeichen vor dem Erreichen von `stop` gelesen, wird das `failbit` gesetzt.

```
int charfeld_eingabe(std::istream& in, char* str, int n, char stop='\n')
{
  in.getline(str, n, ende); // stop wird verschluckt
  in.getline(str, n);       // stop = '\n'
  in.get(str, n, ende);     // stop wird nicht gelesen
  in.get(str, n);           // geht schief, wenn schon bei '\n'
  return in.gcount();       // Anzahl zuletzt gelesener Zeichen
}
```

**Ein- und Ausgabeströme** (`std::iostream`) entstehen durch Mehrfachvererbung und beherrschen sowohl Eingabe- (`std::istream`) und Ausgabemethoden (`std::ostream`).

**Der Stromzustand** von Eingabe- und Ausgabeströmen ist erfragbar:

```
std::ios_base::iostate zustand(std::istream& s) // oder auch std::ostream
{
  if (s.good()) cout << "gut";
  if (s.eof() ) cout << "noch gut, aber am Dateiende";
  if (s.fail()) cout << "Aktion schiefgegangen,"
                        " Strom noch verwendbar";
  if (s.bad() ) cout << "Strom total durcheinander";
  return s.rdstate();
}
```

Er wird intern durch Flags `goodbit, badbit, eofbit, failbit` aus der Basisklasse `std::ios_base` repräsentiert, kann mit `rdstate()` abgefragt und mit `clear()` gelöscht werden. `clear(state)` überschreibt den Zustand komplett, `setstate(bit)` setzt einzelne Bits zusätzlich.[62]

Ein- und Ausgabe-Operatoren selbst definierter Typen sollten so geschrieben werden, dass Variablen beim Einlesen nicht in einen undefinierten Zustand geraten können und der einzulesende Wert erst geändert wird, wenn die Eingabe komplett erfolgreich war. Ein guter Zustand ist Voraussetzung, aber keine Garantie für den Erfolg weiterer Aktionen mit dem Strom. Bei nicht gutem Zustand haben Ausgaben und Eingaben keine Wirkung.[63] Zumeist genügt eine Abfrage:

```
  if (is >> x >> y) { /* Erfolg ... */ }
```

**Zeichenkettenströme** aus `<sstream>` lesen aus und schreiben in `std::string`s:

```
#include <sstream>

std::string stromkapselung(std::string text)
{
  std::istringstream in{text};
  std::ostringstream out;
  std::string tmp;
  while (in >> tmp)
  {
    out << tmp << '\n'; // jedes Wort auf eine Zeile
  }
  return out.str();     // verwandelt Inhalt wieder in string
}
```

Die Methode `str()` liefert eine Kopie des Strominhalts.

[62] Die Namensgebung ist hier teilweise irreführend.

[63] Unter diesem Aspekt sind die Statusabfragen kaum von Nutzen. Die Abfragen erlauben aber in bestimmten, seltenen Situationen, den Stromnzustand zu reparieren.

**Dateiströme** aus `<fstream>` arbeiten auf Dateien:

```
#include <fstream>

void Datei_Ein_und_Ausgabe()
{
  std::ifstream in{"input.txt"};
  std::ofstream out;
  std::fstream fs{"datei.dat",
    std::ios_base::in|std::ios_base::out|std::ios_base::binary};
  out.open("output.txt");
  if (out) { out << "Ich war hier!"; }
  out.close(); // schreibt Daten, schliesst Datei; nicht erforderlich
}              // denn offene Streams schliessen automatisch
```

Die Konstruktoren und `open()` können ein zweites Argument übernehmen, dessen Werte

| | |
|---|---|
| `in` | zum Lesen öffnen |
| `out` | zum Schreiben öffnen |
| `binary` | als Binärdatei statt als Textdatei öffnen |
| `ate` | gleich ans Ende gehen („at end“) |
| `app` | Daten anhängen |
| `trunc` | Dateiinhalt löschen |

in `std::ios_base` definiert sind und verodert werden können. *Textdateien* werden als Folgen von Zeichen aufgefasst, die aber keine 1-zu-1-Entsprechung zu den Zeichenfolgen auf externen Geräten haben müssen. So erfolgen systemabhängig (Windows) bestimmte Zeichenumwandlungen wie von `'\n'` zu `CR/LF`. *Binärdateien* führen keine Zeichenumwandlungen durch. Auch `EOF` kann mitten in einer Binärdatei vorkommen.

**Ströme mit wahlfreiem Zugriff** können die Lage der aktuellen Lese- oder Schreibposition im Strom erfragen und ändern. Dies ist vor allem für Binärdateien wichtig.

```
long size(std::istream& is)
{
  long pos = is.tellg();           // "get"-Position merken
  is.seekg(0, std::ios_base::beg); // an den Anfang gehen
  long beg = is.tellg();
  is.seekg(0, std::ios_base::end); // ans Ende
  long end = is.tellg();
  is.seekg(pos);                   // Zustand wiederherstellen
  return end - beg;
}
```

Die Position kann absolut vom Anfang oder relativ zu Anfang (`ios_base::beg`), Ende (`ios_base::end`) bzw. aktueller Position im Strom (`ios_base::cur`) gesetzt werden:

```
is.seekg(-8, std::ios_base::cur); // acht Byte (ein double?) zurueck
```

Bei Ausgabeströmen funktionieren `tellp()` und `seekp()` (p für „put“) entsprechend.

### C.13.2 Formatierung

*Die Verwendung von Flags zur Kontrolle eines Stream-Zustands ist jedenfalls eher eine Studie über Implementierungstechniken als über Schnittstellen-Design.*
– Bjarne Stroustrup

Das Verhalten der Eingabe-Ströme und das Aussehen der Ausgaben (*Formatierung*) wird durch Schalter (*flags*) in der Basisklasse `std::ios_base` gesteuert.[64] Der Schalter `boolalpha` bewirkt, dass statt `0` oder `1` je nach Ländereinstellung (*locale*) `true` und `false` (Standard englisch) oder `wahr` und `falsch` (deutsch) ausgegeben wird. Ist `skipws` gesetzt (Standard), werden führende Whitespaces bei der Eingabe verschluckt. Die Optionen werden verodert und haben bleibende Wirkung.

```
void formatierung(std::istream& in)
{
  long alt = in.flags(); // alte Schalter erfragen
  long neu = std::ios_base::skipws;
  in.flags( alt | neu ); // alle Schalter neu setzen
  in.setf( neu );        // einzelnes Flag setzen
  in.unsetf( neu );      // einzelnes Flag entfernen
}
```

**Ganzzahlen** sind in drei Zahlensystemen darstellbar:

```
void ganzzahlbasis(std::ostream& out, int i)
{
  out.setf(std::ios_base::dec, std::ios_base::basefield);
  out << i << ' ';                          // dezimal (Standard)
  out.setf(std::ios_base::oct, std::ios_base::basefield);
  out << i << ' ';                          // oktal
  out.setf(std::ios_base::hex, std::ios_base::basefield);
  out << i;                                 // hexadezimal
}
```

gibt `25 31 19` aus. Die Methode `setf(flag, mask)` setzt alle Flags in `mask` zurück und setzt `flag` neu, wirkt also wie `flags(flags() & ~mask | flag&mask)` und verhindert, dass zwei widersprechende Schalter gleichzeitig gesetzt werden. Wurde zusätzlich `showbase` angeschaltet, wird auch die Zahlenbasis mit angezeigt: `25 031 0x19`. Entprechend der eingestellten Zahlenbasis werden auch Eingaben interpretiert:

```
in.setf(std::ios_base::oct, std::ios_base::basefield);
in >> i;
```

Die eingegebenen Ziffern `31` erzeugen den Wert 25 (dezimal).[65]

[64] Häufig wird für `char`-Ströme auf deren abgeleitete Klasse `std::ios` Bezug genommen.
[65] Programmierer verwechseln immer Weihnachten mit Helloween, weil OCT 31 == DEC 25 ist.

**Gleitkommazahlen** zeigen bei Ausgabe den Dezimalpunkt (oder bei deutscher Locale das Komma) nur, wenn `showpoint` gesetzt wird. Die Zahl der Stellen hinter dem Komma kann abgefragt und neu gesetzt werden:

```
void mehrdezimalen(std::ostream& out, int nachkommastellen)
{
  out.precision( nachkommastellen );  // setzen
  nachkommastellen = out.precision(); // abfragen
}
```

Mit der Standard-Einstellung `0` wird die Ausgabe auf 6 Nachkommastellen gerundet, nicht abgeschnitten. Große oder sehr kleine Zahlen werden in der Exponentialschreibweise ausgegeben, z. B. `1.602e-19`. Dieses Format lässt sich für alle Ausgaben mit `scientific` erzwingen oder mit `fixed` unterdrücken.

```
out.setf(std::ios_base::scientific, std::ios_base::floatfield); // setzen
out.unsetf(std::ios_base::scientific | std::ios_base::fixed); // loeschen
out.setf(0, std::ios_base::floatfield);                  // auch loeschen
```

**Positive Vorzeichen** werden bei gesetztem `showpos` gezeigt, durch `uppercase` erscheint `X` in hex-Werten und `E` im scientific-Format groß (Standard: aus für beide).

**Die Ausgabebreite** ist nur für die unmittelbar folgende Ausgabe änderbar, danach wird sie wieder auf den Standardwert `0` gesetzt:[66]

```
out.width(n);  // Ausgabeweite festlegen
n = out.width(); // abfragen
```

Bei zu knappem Platz wird weitergeschrieben[67], bei größerer Breite werden mit

```
char ch = out.fill(); // Standard: Leerzeichen
out.fill( '_' );      // anderes Zeichen einsetzen
```

gesteuerte Füllzeichen eingefügt. Die Ausrichtung erfolgt nach Einstellungen wie

```
out.setf(std::ios_base::internal, std::ios_base::adjustfield);
```

im `adjustfield` und bewirkt für `-1.23456789` die Ausgabe

```
          // width(8), precision(2)
-1.23___  // left (Standard)
___-1.23  // right
-___1.23  // internal
```

`internal` fügt Füllzeichen zwischen Vorzeichen und Zahlwert ein.

[66] Alle anderen Einstellungen sind dauerhaft.
[67] Lieber hässlich richtig als schön falsch.

Tabelle 12: Parameterlose `iostream`-Manipulatoren und ihre Wirkung.

| | | | |
|---|---|---|---|
| `flush` | | | leert den Puffer |
| `endl` | | | fügt `'\n'` ein und leert den Puffer |
| `ends` | | | fügt `'\0'` ein und leert den Puffer |
| `boolalpha` | `noboolalpha` | | Wahrheitswerte als Wort oder Zahl |
| `showbase` | `noshowbase` | | Anzeige der Zahlenbasis `0...` oder `0x...` |
| `showpoint` | `noshowpoint` | | Komma bei ganzen Gleitkommazahlen |
| `showpos` | `noshowpos` | | positives Vorzeichen zeigen |
| `skipws` | `noskipws` | `ws` | Leerzeichen vor Eingabe verschlucken |
| `uppercase` | `nouppercase` | | `X` und `E` groß |
| `dec` | `hex` | `oct` | Wechsel der Zahlenbasis |
| `fixed` | `scientific` | | Darstellung der Gleitkommazahlen |
| `left` | `internal` | `right` | Ausrichtung in breiten Feldern |

Tabelle 13: `iostream`-Manipulatoren mit Parametern (`<iomanip>` ist einzubinden).

| | |
|---|---|
| `setbase(base)` | Wechsel der Zahlenbasis (8, 10, 16) |
| `setprecision(n)` | Genauigkeit von Gleitkommazahlen |
| `setw(n)` | Ausgabefeldbreite |
| `setfill(c)` | Füllzeichen |
| `setiosflags(f)` | Setzen und |
| `resetiosflags(f)` | Rücksetzen von Flags |

**Manipulatoren** (Tabellen 12–13) bieten elegantere Schreibweisen an. Manipulatoren mit Argumenten sind in `<iomanip>` definiert (Tabelle 13).

```
int i = 123;
double d = 123.456789;
std::cout << std::hex << i
          << std::dec << std::showpos << ' ' << i << '\n';
std::cout << std::setw(12) << std::right    << i << '\n'
          << std::setw(12) << std::internal << i << '\n'
          << std::setw(12) << std::setprecision(3) << d << '\n'
          << std::setw(12) << std::scientific      << d << '\n';
```

Der Methoden-Stil wirkt dagegen grauenhaft (hier nur für die Zeilen 3 und 4):

```
std::cout.setf(std::ios_base::hex, std::ios_base::basefield);
std::cout << i;
std::cout.setf(std::ios_base::dec, std::ios_base::basefield);
std::cout.setf(std::ios_base::showpos);
std::cout << ' ' << i << '\n';
std::cout.flush();
```

**C-Ausgabeformatierung** mit `int sprintf(char* s, const char* fmt, ...)` erlaubt Ausgaben in (ausreichend lange) Puffer-Zeichenketten `s[]` im Hauptspeicher:

```
#include <cstdio>

std::string C_ausgabe(char c = 'A', int i = 123, double d = 1.23)
{
  char buf[1000]; // genug?
  std::sprintf(buf, "char: %c int: %d float: %f\n", c, i, f);
  std::sprintf(buf, "Anteil = %07.3lf%%\n", f);    // Anteil = 01.230%
  std::sprintf(buf, ":%10.5s:\n", "Hallo, Welt");  // :     Hallo:
  int breite = 10, anzahl = 5;
  std::sprintf(buf, ":%-*.*s:\n", breite, anzahl, "Hallo, Welt");
  return buf;                                      // :Hallo     :
}
```

Die durch `%` eingeleiteten kompakten *Formatieranweisungen* in der Formatzeichenkette `fmt` steuern die Darstellung der nachfolgenden Argumente.[68] Sie enthalten optional

- ein `-` (bei linksbündiger Ausgabe), `+` (Zahl immer mit Vorzeichen) oder Leerzeichen (vor positiver Zahl),
- ein `#` zur „alternativen“ Darstellung (Gleitkommazahlen immer mit Dezimalpunkt, immer auffüllende Nullen, Anzeige der Oktal-/Hexadezimalbasis)
- eine Zahl, welche die Feldbreite der Ausgabe bestimmt (bei vorangehender Null werden Zahlen links mit Nullen aufgefüllt),
- ein Dezimalpunkt, dem eine Zahl für die Genauigkeit der Ausgabe folgt (Nachkommaziffern einer Gleitkommazahl oder Anzahl der Zeichen einer Zeichenfolge),
- ein `*` anstelle von Feldbreite oder Genauigkeit wertet das nächste Argument als diese Angabe aus,

und zwingend einen Buchstaben für den Datentyp des Arguments. Ein vorausgehendes `h`, `l` oder `L` weist auf `short`, `long` und `double` bzw. `long double` hin:

| | |
|---|---|
| `c` | Zeichen |
| `d` | Ganzzahl (dezimal) |
| `e` oder `E` | Gleitkommazahl mit Exponent nach `e` oder `E` |
| `f` | Gleitkommazahl |
| `g` oder `G` | der kürzere von `f` und `e` oder `E` |
| `o` | Ganzzahl (oktal) |
| `n` | bisher geschriebene Zeichen (erfordert `int*`-Argument) |
| `p` | Zeiger |
| `s` | Zeichenkette |
| `u` | vorzeichenlose Ganzzahl |
| `x` oder `X` | Ganzzahl (hexadezimal), Ziffern >9 als `a-f` oder `A-F` |

[68] Irrtümer bei der Zuordnung von Format und Argumenten führen *ohne Warnung* zu Laufzeitfehlern. Die Ausgabe eines Prozentzeichens erfolgt mit `%%`.

```
verarbeite(*iterator)
                |
[0|1|2|3|4|5|9|8|7|6|5|4]
 ^           ^           ^
 |           |  -->      |
 first     ++iterator    last
```

Abbildung 4: Daten-Sequenz als halboffener (Speicher-)Bereich.

## C.14 Datencontainer

### C.14.1 Verarbeitung von Datenfolgen und Datenhaltung

Häufig müssen Mengen gleichartiger Daten (*Sequenzen*) durchlaufen (*iteriert*[69]) und in geeigneter Weise verarbeitet werden (Abb. 4). Die Organisation der Datenablage ist zur Lösung der Aufgabe unwichtig, aber mitunter entscheidend für die Effizienz des Ablaufs — eine ermüdende und fehlerträchtige Routineaufgabe. Felder `arr[]` werden als Zeiger verwaltet (mit allen Folgen), besitzen keine Kopiersemantik und ihre Behandlung (Kopie, Freigabe, Prüfung der Grenzen) liegt in der Verantwortung des Programmierers:

```
typedef int T;                          // irgendein Typ ...
T arr[] = { 0,1,2,3,4,5,9,8,7,6,5,4 };
T *first = arr;                         // "Zeiger" auf erstes Element
T *last  = arr+sizeof(arr)/sizeof(T); // "Zeiger" hinter letztes Element
```

Die Daten bilden einen *Bereich*[70] `[first,last)`. Die linke Grenze gehört dazu, die rechte Grenze nicht. Die Bereichsgrenzen `first` und `last` sind Start- und Vergleichswerte für *Iteratoren*. Diese werden dereferenziert (`*iterator`), um auf Elemente zuzugreifen und weitergesetzt (`++iterator` oder `iterator++`), um durch die Sequenz zu wandern:

```
for (auto iterator = first; iterator != last; ++iterator)
  std::cout << *iterator << '\n';  // Elemente von [first,last) ausgeben
```

**Iteratoren** sind ein allgemeines Konzept, das nicht als Zeiger implementiert sein muss. Es unterliegt aber ähnlichen Einschränkungen und Vereinbarungen wie (sinnvolle) Zeiger, sonst ist das Verhalten bestenfalls undefiniert, möglicherweise verheerend:

- Iteratoren müssen auf eine gültige Datensequenz zeigen (keine „dangling pointer“). Vergleiche `== !=`, evtl. auch `<`, haben nur im Bezug auf den Bereich einen Sinn.
- `last` steht unmittelbar „hinter“ dem letzten gültigen Element. Die Dereferenzierung `*last` ist unsinnig.
- `last` darf nicht „vor“ `first` stehen, sonst wird das Schleifenende nicht erreicht. Bei `first==last` ist der Bereich leer.

[69] iterare = lat.: wiederholen.
[70] In mathematischer Sprechweise ein (nach rechts) halboffenes Intervall diskreter Elemente.

**Container** der Standardbibliothek

- organisieren die datentyp-parametrisierte Datenhaltung von Bereichen,
- verfolgen unterschiedliche Speicherstrategien (Feld, Liste, Baum, Hashtabelle),
- abstrahieren von Implementierungsdetails und fehlerträchtigen Routineaufgaben, übernehmen die Speicherverwaltung,
- lassen sich mit Iteratoren an *Algorithmen* koppeln und sind durch einheitliche Schnittstellen bei wechselnden Anforderungen leichter austauschbar.

Container verändern auch das Herangehen an eine Aufgabe. Der Ansatz

```
struct Eintrag {
  std::string;
  int  nummer;
};
Eintrag telefonbuch[1000];
```

verschwendet Speicher oder reserviert zu wenig.[71] Besser wäre

```
std::vector<Eintrag> telefonbuch;
telefonbuch.push_back({"Meier", 6135});
```

Bei häufigem Einfügen oder Löschen von Teilbereichen bietet eine Liste Vorteile:

```
std::list<Eintrag> telefonbuch;
```

Die Suche nach `name` in Feldern und Listen ist mühsam. Näher am Nachschlageproblem eines Telefonbuchs ist ein *assoziatives Feld*, das *Werte* `nummer` nach einem *Schlüssel* `name` sortiert und schnell findet:

```
std::map<std::string, int> telefonbuch;
// ...
telefonbuch["Meier"] = 6135;  // "Meier" neu eintragen
telefonbuch["Meier"] = 6134;  // "Meier"s Nummer wechseln
// ...
std::cout << telefonbuch["Meier"] << '\n'; // "Meier"s Nummer: 6134
std::cout << telefonbuch["Meyer"] << '\n'; // 0 (bisher nicht vorhanden)
```

Noch schneller kann man in einer Hashtabelle nachschlagen:

```
std::unordered_map<std::string, int> telefonbuch;
```

Container mit gleicher Schnittstelle lassen sich schnell austauschen:

```
using Container = std::vector<T>;  // oder besser std::list<T> ?
Container c;
```

[71] Beispiel aus B. Stroustrup: C++, 3rd edn.

### C.14.2 Containerarten

Die Standardbibliothek (Tab. 7) bietet eine Fülle von Containern (Abb. 5).

**Sequentielle Container** halten übernommene Elemente in vorgegebener Folge:

```
auto v = std::vector{1,2,3,4}; // Initialisiererliste mit int-Werten
```

Jede Containerart hat ihre Stärken. `std::deque<T>` kann beidseitig wachsen Listen lassen sich gut umordnen. `std::array<T, N>` benötigt keinen dynamischen Speicher.

```
std::array<T, N>      seq0;              // Feld mit genau N Elementen
std::vector<T>        seq1(first, last); // dynamisches Feld
std::deque<T>         seq2(first, last); // "double-ended queue"
std::forward_list<T>  seq3(first, last); // einfach verkettete Liste
std::list<T>          seq4(first, last); // doppelt verkettete Liste
```

**Assoziative Container** verbinden (*assoziieren*) Werte mit Schlüsseln und halten die
►C.16.1 Schlüssel mit einem Kriterium `Compare`, wenn nicht angegeben, `std::less<T>`, sortiert:[72]

```
std::set<T, std::greater<>> abwaerts (first, last); // aus <functional>
std::set<T>                 aufwaerts(first, last);
std::map<std::string, std:string> woerterbuch;
```

Werte in `std::set<T>` sind zugleich Schlüssel. Tabellen `std::map<Key, Value>` halten Schlüssel-Wert-Paare. Hier kann der Schlüssel wie ein Feldindex eingesetzt werden. Ist der Schlüssel bisher nicht enthalten, wird er eingefügt. Die Methode `find(key)` liefert den Iterator auf den Eintrag oder `end()`, wenn der Schlüssel nicht gefunden wird.

**Hash-Container** finden Einträge mit `==` und `Hash`-Funktor für den `Key`-Typ:[73]

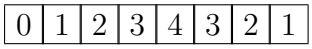
```
struct K { int id; /* ... */ };
bool operator==(K a, K b) { return a.id == b.id; }

struct MyHash {
  size_t operator()(K x) const { return std::hash<int>()(x.id); }
};

#include <functional> // oder std::hash<T> spezialisieren
namespace std {
  template <> struct hash<K> {
    size_t operator()(K x) const { return hash<int>()(x.id); }
  };
}
std::unordered_set<K, MyHash> menge1;
std::unordered_set<K>         menge2;
```

[72] Suche und Zugriff auf Schlüssel sind im balancierten Baum schneller als bei Sequenz-Containern. In `std::multiset<T>` und `std::multimap<Key, Value>` dürfen gleiche Schlüssel mehrfach vorkommen.

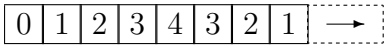
[73] Hier kommt es nicht auf die Reihenfolge an, sondern auf Schnelligkeit.

*Sequentielle Container*:

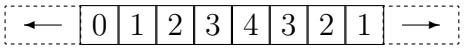
`array<T,N>`



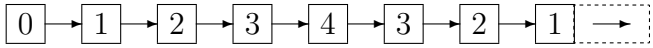

`vector<T>`



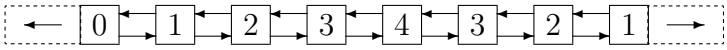

`deque<T>`




`forward_list<T>`




`list<T>`




*Assoziative und Hash-Container*:

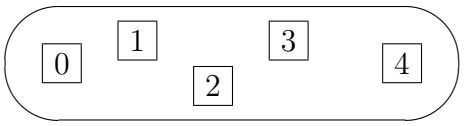
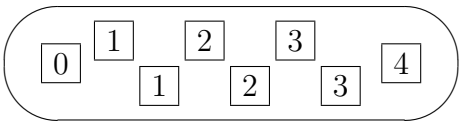
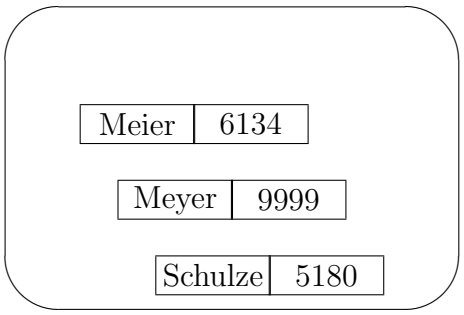
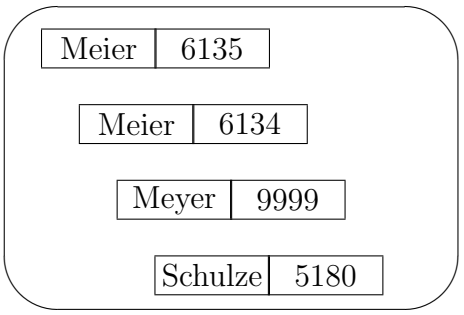
| ohne mehrfache gleiche Schlüssel | gleiche Schlüssel mehrfach möglich |
|---|---|
| `set<T, Compare>`<br>`unordered_set<T, Hash>` | `multiset<T, Compare>`<br>`unordered_multiset<T, Hash>` |
|  |  |
| `map<Key,Value,Compare>`<br>`unordered_map<Key,Value,Hash>` | `multimap<Key,Value,Compare>`<br>`unordered_multimap<Key,Value,Hash>` |
|  |  |



Abbildung 5: Container der Standardbibliothek.

### C.14.3 Allgemeine Containereigenschaften

**Container als Eigentümer** kopieren die Inhalte eines gleichartigen Containers oder eines Bereiches `[first,last)`[74] und sind für deren Freigabe verantwortlich:

```
void container_demo(const Container& c)
{
  Container c0, c1{c}, c2 = c;      // leer bzw. Kopie
  Container c3(first, last);        // Kopie aus allgem. Bereich
  c0 = c3;                          // Zuweisung
```

**Speicherabfragen** liefern aktuelle und maximal mögliche Elementanzahl:[75]

```
  if (c.empty()) std::cout << "Der Container ist leer.\n";
  std::cout << c.size() << " von max. " << c.max_size() << " Daten.\n";
```

**Lexikographische Vergleiche** mit `== != ` bzw. `< <= >= >` von Containern gleichen Typs erfolgen durch Vergleich ihrer Elemente, wenn für die Elemente `operator<()` als totale Ordnung und `operator==()` als Äquivalenzrelation definiert sind.

**Vertauschungen** sind als Methode und globale Funktion erlaubt:

```
  if (c1 != c2) c1.swap(c2);
  std::swap(c1, c2);                // ruft c1.swap(c2);
}
```

**Einfügen und Entfernen** verschieben Folgeelemente so, dass keine Lücke entsteht:[76]

```
void insert_demo(Container c)
{
  c.insert(c.end(), -1);            // ein Wert -1 ans Ende
  c.insert(c.begin(), first, last); // ganzen Bereich nach vorn
}

void erase_demo(Container c)
{
  c.erase(c.begin());   // loescht ein Element an gegebener Position
  c.erase(c.begin(), c.end());      // loescht ganzen Bereich
  c.clear();                        // loescht alle Elemente
}
```

[74] `std::array<T, N>` kann keine Bereiche kopieren.

[75] Für `std::forward_list<T>` ist `size()` nicht implementiert, da zu teuer.

[76] Die Iteratorposition muss im Containerbereich liegen. Nicht möglich bei `std::array<T,N>`; bei `std::forward_list<T>` heißen die Methoden `insert_after()` und `erase_after()`.

Einzufügende Objekte können mit `emplace(pos, args)` direkt im Container erschaffen werden; dazu werden die Konstruktorargumente an den Container durchgereicht:

```
std::vector<Punkt> v;
v.emplace(v.end(), x, y);  // statt v.insert(v.end(), Punkt(x, y));
```

### C.14.4 Spezielle Containeroperationen

**Indexoperatoren** `[]` und `at(index)` mit Gültigkeitsprüfung sind für die Container `std::array<T,N>`, `std::vector<T>`, `std::deque<T>`, `std::map<Key, Value>` erlaubt.[77]

**Stapel-Methoden** sind auf vielen Containern durchführbar:[78]

| | |
|---|---|
| `emplace_back(args)` | schafft bzw. |
| `push_back(value)` | fügt einen Wert am Ende ein, |
| `emplace_front(args)` | ... |
| `push_front(value)` | (dasselbe am Anfang), |
| `pop_back()` | entfernt das letzte Element bzw. |
| `pop_front()` | das erste Element, |
| `front()` | gibt Zugriff auf erstes bzw. |
| `back()` | letztes Element. |

**Listentypische Methoden** umfassen[79]

| | |
|---|---|
| `merge(list2)` | Verschmelzen sortierter Listen, |
| `splice(pos, list2)` | Umhängen von Bereichen, |
| `splice(pos, first, last)` | |
| `sort(compare)` | Sortieren mit `std::less<T>()` o.a. Kriterium, |
| `remove(value)` | Entfernen aller Vorkommen eines Wertes bzw. |
| `unique()` | benachbarter Dubletten. |

**Methoden von assoziativen Containern** sind auch bei Hash-Containern erlaubt:

| | |
|---|---|
| `emplace(args)` | schafft bzw. |
| `insert(value)` | fügt Wert an passender Stelle ein; |
| `emplace_hint(pos, args)` | können schneller sein, |
| `insert_hint(pos, value)` | wenn die passende Stelle angegeben wurde, |
| `find(key)` | liefert Iterator auf Element bzw. |
| `count(key)` | Anzahl der Elemente mit gegebenem Schlüssel. |

**Hash-Container-Methoden** werden im Standard erwähnt, aber nicht erläutert:[80]

| | |
|---|---|
| `bucket_count()` | `load_factor()` |
| `bucket_size(n)` | `max_load_factor()` |
| `bucket(key)` | |

[77] Auf Listen wäre zum wahlfreien Zugriff ein (unvollständiger) Durchlauf erforderlich.

[78] Nur kostengünstige Operationen werden implementiert: Ein `insert(pos, value)` am Anfang eines Vektors erfordert das elementweise Verschieben/Kopieren der Folgeelemente.

[79] Bei `std::forward_list<T>` erfordern die Methoden `splice_after(...)` offene Bereiche.

[80] Siehe http://www.boost.org/doc/libs/1_48_0/doc/html/unordered.html.

Tabelle 14: Laufzeitverhalten einiger Algorithmen

| | | |
|---|---|---|
| | | billig |
| $O(1)$ | konstant | Element `e` ans Ende anfügen: `l.push_back(e)` |
| $O(\log n)$ | logarithmisch | Suche im Baum (assoziative Container) |
| $O(n)$ | linear | Suche in Sequenz, Einfügen vorn in vector |
| $O(n \log n)$ | | `quicksort()` |
| $O(n^2)$ | quadratisch | `bubblesort()` |
| $O(n^3)$ | kubisch | Multiplikation n-reihiger Matrizen |
| $O(\exp n)$ | exponentiell | Primzahlsuche? |
| $O(n!)$ | kombinatorisch | `permutation_sort()` — die Prolog-Lösung |
| | | nicht beherrschbar teuer |

### C.14.5 Kosten von Operationen

*One of the cultural barriers that separates computer scientists from "regular" scientists and engineers is a differing point of view on whether a 30% or 50% loss of speed is worth worrying about. In many real-time or state-of-the-art scientific applications, such a loss is catastrophic. The practical scientist is trying to solve tomorrow's problems with yesterday's computer; the computer scientist, we think, often has it the other way around.*
– Numerical Recipes in C

Für den Einsatz ist ihr Laufzeitbedarf (Tab. 14) bei wachsender Datenmenge $n$ wichtig.[81] Die Containermethoden zeigen folgendes Zeitverhalten:

| | `vector` | `deque` | `list` | `set/map` | `unordered_set/map` |
|---|---|---|---|---|---|
| Indexzugriff `[]` | $O(1)$ | $O(1)$ | - | $O(\log n)$ | $O(1)+$ |
| Listenoperationen | $O(n)+$ | $O(n)$ | $O(1)$ | $O(\log n)+$ | $O(1)+$ |
| Operationen vorn | - | $O(1)$ | $O(1)$ | - | - |
| Operationen hinten | $O(1)+$ | $O(1)$ | $O(1)$ | - | - |
| Iteratoren | Ran | Ran | Bi | Bi | Bi |

| | |
|---|---|
| - | nicht implementiert (wegen schlechten Zeitverhaltens) |
| + | durchschnittliches Verhalten, gelegentlich Zusatzaufwand |
| Bi | Bidirektional-Iteratoren |
| Ran | Random-Access-Iteratoren |

Keine der aufgeführten Container-Operationen ist $O(n^2)$ oder teurer. Iteratoroperationen wird konstantes Zeitverhalten zugesichert.

[81] Die Schreibweise $O(f(n))$ gibt den (asymptotischen) Zeitverbrauch $t(n) = c \cdot f(n)$ bis auf einen Vorfaktor $c$ an.

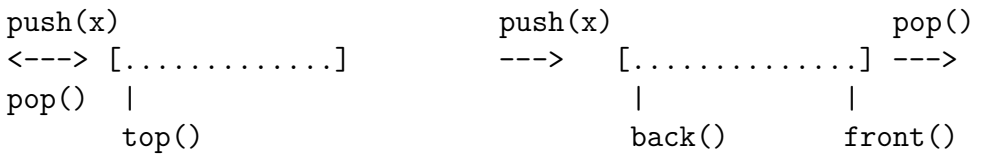



Abbildung 6: Stapel und Warteschlangen.

### C.14.6 Container-Adapter

*Stapel* `std::stack<T>` und *Warteschlangen* `std::queue<T>` (Abb. 6) sind Container-*Adapter*. Ihre Funktion erfüllt ein beliebiger sequentieller Container, der vom Adapter umschlossen wird. *Prioritätswarteschlangen* ordnen Elemente mit *großen* Werten so ein, dass sie *zuerst* wieder herauskommen. Die Angabe des Containers und des Vergleichskriteriums kann entfallen — angegeben sind hier die Standardfacetten:

```
#include <stack>
#include <queue>

std::stack<T, std::deque<T>> s;
std::queue<T, std::deque<T>> q;
std::priority_queue<T, std::vector<T>, std::less<T>> p;
```

Die Adapter besitzen neben Konstruktor und Vergleich nur wenige Methoden:

| | |
|---|---|
| `full()` | Adapter ist voll, |
| `empty()` | Adapter ist leer, |
| `push(x)` | fügt x hinten bzw. oben ein, |
| `pop()` | löscht erstes Element, |
| `top()` | Referenz auf oberstes Element im Stapel, |
| `front()` | auf erstes / |
| `back()` | letztes Element der Warteschlange. |

```
void adapter_demo(std::stack<T> s, std::queue<T> q)
{
  while (!s.empty() && !q.full())
  {
    q.push(s.top());
    s.pop();
  }
  while (!q.empty())
  {
    std::cout << q.front() << ' ';
    q.pop();
  }
  std::cout << '\n';
}
```

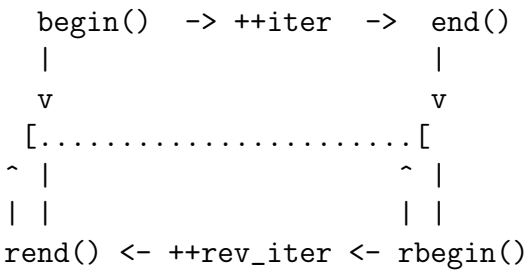



Abbildung 7: Vorwärts- und Rückwärts-Iteratoren.

### C.14.7 Iteratoren

Iteratoren erlauben die containerunabhängige Formulierung von Algorithmen:

```
template <class Iterator>
void print_range(std::string header, Iterator first, Iterator last)
{
  std::cout << header;
  for (auto i = first; i != last; ++i)
    std::cout << *i << ' ';
  std::cout << '\n';
}
```

Container definieren selbst ihre Iteratoren zum Vorwärts- und Rückwärts-Durchlaufen und geben ihre eigenen aktuellen Bereichsgrenzen an:

| nicht konstante / | konstante Container | Elementbereich |
|---|---|---|
| `iterator` | `const_iterator` | `[begin(), end())` |
| `reverse_iterator` | `const_reverse_iterator` | `[rbegin(), rend())` |

```
void iterator_demo(Container c)
{
  Container::iterator         first  = c.begin(); // oder begin(c)
  Container::iterator         last   = c.end();   // oder end(c)
  Container::reverse_iterator rfirst = c.rbegin();
  Container::reverse_iterator rlast  = c.rend();

  print_range("Vorwaerts:   ", first,  last );
  print_range("Rueckwaerts: ", rfirst, rlast);
}
```

*Forward-Iteratoren* beherrschen zumindest `*iter` und `++iter` bzw. `iter++` und lassen sich mit `==` und `!=` vergleichen. Auf *Bidirektional-Iteratoren* können zusätzlich `--i` bzw. `i--` angewendet werden, auf *Random-Access-Iteratoren* zusätzlich `i[n]` und `*(i+n)`. Bei *Rückwärts-Iteratoren* ist die Wirkung von `++` und `--` vertauscht: `rev_iter++` bewegt den `reverse_iterator` zum vorhergehenden Element. Wegen der Halboffenheit der Bereiche ist ihr Bezug um ein Element verschoben (Abb. 7). Mit `rev_iter.base()` kann der zugehörige Vorwärts-Iterator zurückgewonnen werden.

### C.14.8 Iterator-Adapter

Iterator-Adapter aus `<iterator>` vereinheitlichen die Schreibweise für Einfügen, Ein- und Ausgaben trotz unterschiedlicher Quellen und Ziele: ►C.15

```
while (first != last) *ins++ = *first++; // Kopie des Bereichs einfuegen
while (in != ende)    *out++ = *in++;    // Kopie von Eingabe zur Ausgabe
```

**Einfüge-Iteratoren** rufen eine Einfügeoperation eines Containers auf:

```
void insert_iteration(Container& c, T wert)
{
  auto pos = begin(c); // gueltige Position: c.begin()
  std::insert_iterator<Container>     ins = inserter(c, pos);
  std::front_insert_iterator<Container> j = front_inserter(c);
  std::back_insert_iterator <Container> k = back_inserter (c);
  *ins = wert;   // c.insert(pos, wert);
  *j = wert;     // c.push_front(wert) oder c.insert(begin(c), wert)
  *k = wert;     // c.push_back (wert) oder c.insert(end(c), wert)
  ins++;         // wirkungslos
  j++;
  k++;
}
```

**Ausgabe-Strom-Iteratoren** geben Werte und Trennzeichenfolgen in einen Strom aus:

```
void ausgabe_iteration(std::ostream& os, T wert)
{
  std::ostream_iterator<T, char> out{os, " "}; // Standardtrenner: " "
  *out = wert;   // os << wert << " ";
  out++;         // wirkungslos
}
```

**Eingabe-Strom-Iteratoren** lesen Werte aus dem Strom und behalten diese. Beim Inkrement wird der nächste Wert eingelesen.

```
void eingabe_iteration(std::istream& is, T& wert)
{
  std::istream_iterator<T, char> in{is}; // liest ersten Wert: is >> tmp;
  std::istream_iterator<T, char> ende;   // Ende des Stroms

  if (in != ende) // erfolgreich gelesen ?
  {
    wert = *in;  // Wert liefern:     wert = tmp;
    in++;        // neuen Wert lesen: is >> tmp;
  }
}
```

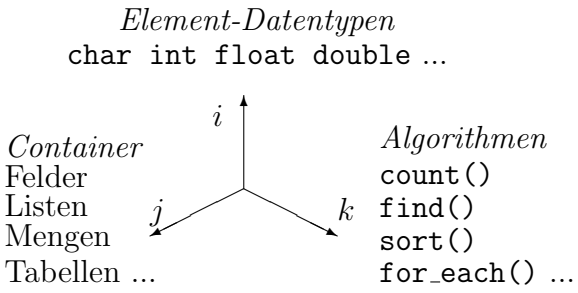



Abbildung 8: Komponentenraum der Standardbibliothek.

## C.15 Algorithmen

### C.15.1 Überblick

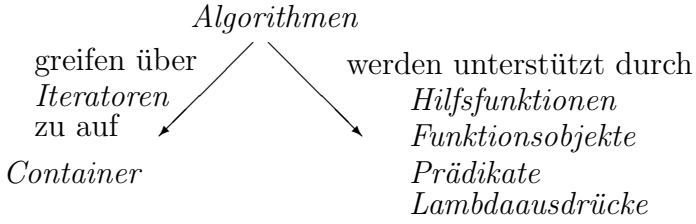
**Komplexe Abläufe** lassen sich nicht auf einen Blick erfassen:

```
result = first;
if (first != last)
  while (++first != last)
  {
    if (*result < *first) result = first;
  }
std::cout << *result << '\n';
```

Verständlicher und weniger fehleranfällig ist die gleichwertige Anweisung:

```
std::cout << *std::max_element(first, last) << '\n';
```

Die von Alexander Stepanov entworfene Standard Template Library (STL) setzt eine alte Idee von Niklaus Wirth um: Programme = Datenstrukturen + Algorithmen.




Ohne Templates scheitert die Idee an der Komplexität der Aufgabe (Abb. 8)[82]: Bei $i$ unterschiedlichen Datentypen, $j$ verschiedenen Arten von Datenbehältern und $k$ Prozeduren erfordert das $i * j * k$ verschiedene Definitionen. Keine Bibliothek wäre vollständig, jeder neue Datentyp oder Algorithmus würde eine Welle von Ergänzungen erforderlich machen. Die in die Standardbibliothek von C++ integrierte STL[83] reduziert die Komplexität durch *generische Programmierung* über Iteratoren entkoppelte Komponenten:

[82] Nach: Johannes Weidl, *STL Tutorial*, TU Wien.
[83] Sie wird mittlerweile auch nicht mehr STL genannt.

- datentypunabhängige Container $i * j \to j$

```
std::vector<int> v;
std::list<float> l;
```

- und containertypunabhängige Algorithmen auf Bereichen $j * k \to j + k$:

```
std::sort(begin(v), end(v));
std::sort(begin(l), end(l));
```

**Zusammenwirken** von Containern, Algorithmen und Iteratoren (Abb. 15) mit Hilfsfunktionen bzw. -objekten macht die Bibliothek leistungsfähig und ausdrucksstark: ▶C.16

```
#include <cmath>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

struct Statistics  // Funktionsobjekt
{
  Statistics() : n{0}, sum{0}, sqr{0} {}
  void operator() (double x) { n++; sum += x; sqr += x*x; }
  int size()       const { return n; }
  double average() const { return n ? sum/n : 0; }
  double deviate() const { return n ? std::sqrt(n*sqr-sum*sum)/n : 0; }
private:
  int n;
  double sum, sqr;
};

int main()
{
  std::istream_iterator<double> first{std::cin}, last;
  std::vector<double> v(first, last);
  auto stat = std::for_each(begin(v), end(v), Statistics{});

  std::cout << stat.size() << " : "
            << stat.average() << " +/- " << stat.deviate() << '\n';
  if (!v.empty())
  {
    auto range = std::minmax_element(begin(v), end(v));
    std::cout << "[ " << *range.first << ", " << *range.second << " ]\n";
  }
}
```

Tabelle 15: Forderungen an Iteratoren in `<algorithm>`

| | | |
|---|---|---|
| In | Eingabeiterator | als rvalue verwendbar: `wert = *in;` |
| Out | Ausgabeiterator | als lvalue verwendbar: `*out = wert;` |
| For | Vorwärtsiterator | les- und schreibbar: `*forw = *forw;` |
| Bi | Bidirektionaliterator | `bi--` möglich |
| Ran | Random-Access-Iterator | `ran+=n` möglich |

### C.15.2 Nichtmodifizierende Algorithmen

**Funktions(objekt)aufruf** `func(*i)` für jedes Element `*i` des Bereichs

```
Function for_each(In first, In last, Function func);
```

gibt das eventuell veränderte Funktionsobjekt zurück. Obwohl alle nichtmodifizierenden Algorithmen keine Änderungen an den Elementen vornehmen, ist dies dem Funktor von `for_each()` ausdrücklich erlaubt.

**Prädikatenlogische Quantoren** prüfen das Zutreffen einer Eigenschaft `pred(*i)` auf mindestens eines (Existenzquantor $\exists x : pred(x)$), auf alle (Allquantor $\forall x : pred(x)$) und auf keines ($\forall x : \neg pred(x) \Leftrightarrow \neg\exists x : pred(x)$) der Elemente im Bereich:

```
bool all_of(In first, In last, Pred pred);
bool any_of(In first, In last, Pred pred);
bool none_of(In first, In last, Pred pred);
```

Bei leerem Bereich liefert `any_of()` das Ergebnis `false`, die anderen Quantoren `true`.

**Zählen der Elemente** mit dem Wert `value` oder mit dem zutreffenden einstelligen Prädikat `pred(*i)` im Bereich `[first,last)` erfolgt durch

```
difference_type count(In first, In last, T value);
difference_type count_if(In first, In last, Pred pred);
```

**Suchalgorithmen** finden die Position `i` des ersten Elements in `[first,last)` mit `*i==value`, `pred(*i)==true` bzw. `pred(*i)==false` oder geben `last` zurück:

```
In find(In first, In last, T value);
In find_if(In first, In last, Pred pred);
In find_if_not(In first, In last, Pred pred);
```

Wertgleichheit oder zutreffendes zweistelliges Prädikat `pred(*i, *i2)` mit einem Element von `[first2,last2)` finden

```
For find_first_of(For first, For last, For2 first2, For2 last2);
For find_first_of(For first, For last,
                  For2 first2, For2 last2, Binary pred);
```

den Beginn der ersten Teilfolge von `[first,last)`, die `[first2,last2)` oder `count` Werten `value` gleicht bzw. bei denen `pred(*i,*i2)` oder `pred(*i,value)` zutrifft:

```
For search(For first, For last, For2 first2, For2 last2);
For search(For first, For last, For2 first2, For2 last2, Binary pred);
For search_n(For first, For last, Size count, T value);
For search_n(For first, For last, Size count, T value, Binary pred);
```

den Beginn der letzten Teilfolge mit derselben Bedingung:

```
For find_end(For first, For last, For2 first2, For2 last2);
For find_end(For first, For last, For2 first2, For2 last2, Binary pred);
```

oder die Position `i` mit `*i==*(i+1)` oder `pred(*i,*(i+1))` (gleiche Nachbarn):

```
For adjacent_find(For first, For last);
For adjacent_find(For first, For last, Binary pred);
```

**Binäre Suche** im sortierten Bereich findet schnell, ob ein Wert enthalten ist:

```
bool binary_search(For first, For last, T value);
bool binary_search(For first, For last, T value, Comp comp);
```

Unter- und Obergrenzen sind die erste bzw. letzte Position, an denen `value` eingesetzt werden kann, ohne die Sortierung des Bereiches `[first,last)` mit `i<j` oder `comp(i,j)` zu zerstören. `equal_range()` liefert beide Grenzen auf einmal:

```
For lower_bound(For first, For last, T value);
For lower_bound(For first, For last, T value, Comp comp);
For upper_bound(For first, For last, T value);
For upper_bound(For first, For last, T value, Comp comp);
pair<For, For> equal_range(For first, For last, T value);
pair<For, For> equal_range(For first, For last, T value, Comp comp);
```

**Minimum und Maximum** zweier Werte oder im Bereich `[first,last)` werden durch Vergleich mit `<` oder einem Kriterium `comp(a,b)` ermittelt:

```
const T& min(const T& a, const T& b);
const T& min(const T& a, const T& b, Comp comp);
const T& max(const T& a, const T& b);
const T& max(const T& a, const T& b, Comp comp);
For min_element(For first, For last);
For min_element(For first, For last, Comp comp);
For max_element(For first, For last);
For max_element(For first, For last, Comp comp);
pair<T,T> minmax(const T& a, const T& b)
pair<T,T> minmax(const T& a, const T& b, Comp comp)
pair<For,For> minmax_element(For first, For last)
pair<For,For> minmax_element(For first, For last, Comp comp)
```

Das Iteratorpaar von `minmax_element()` liefert den weitesten Teilbereich `[min,max)`.

**Vergleiche** auf elementweise Gleichheit und lexikographische Ordnung[84] erfordern im zweiten Bereich mindestens soviel Elemente wie in `[first,last)`:

```
bool equal(For first, For last, For2 first2);
bool equal(For first, For last, For2 first2, Binary pred);
bool lexicographical_compare(In first, In last, In2 first2, In2 last2);
bool lexicographical_compare(In first, In last,
                             In2 first2, In2 last2, Comp comp);
```

Die Positionen, an denen sich zwei Bereiche erstmals unterscheiden, finden

```
pair<In, In2> mismatch(In first, In last, In2 first2);
pair<In, In2> mismatch(In first, In last, In2 first2, Binary pred);
```

### C.15.3 Modifizierende Algorithmen

Sie können Werte ändern und umordnen.

**Vertauschen** lassen sich einzelne Werte, die als Referenz oder durch Zeiger bzw. Iteratoren gegeben sind, als auch Bereiche. Der zweite Bereich muss groß genug sein; das Ende des vertauschten Bereiches wird zurückgegeben:

```
void swap(T& a, T& b);
void iter_swap(Iter a, Iter b);
For2 swap_ranges(For first, For last, For2 first);
```

**Kopieren** erfordert ebenfalls einen ausreichend großen Zielbereich:[85]

```
Out copy(In first, In last, Out result);
Out copy_n(In first, Size n, Out result);
Out copy_if(In first, In last, Out result, Pred pred);
Bi2 copy_backward(Bi first, Bi last, Bi2 result);
```

Ergebniswert ist das neue Ende des Zielbereichs. Verschieben statt Kopieren erledigen

```
Out move(In first, In last, Out result);
Bi2 move_backward(Bi first, Bi last, Bi2 result);
```

**Ausfüllen** eines (ausreichenden) Bereichs mit neuen Werten (Bsp. S. 104) realisieren

```
void fill(Out first, Out last, T value);
void fill_n(Out first, Size n, T value);
void generate(Out first, Out last, Func generator_obj);
void generate_n(Out first, Size n, Func generator_obj);
```

[84] `true`, sobald beim paarweisen Vergleich ein Element des ersten Bereichs kleiner ist, sonst `false`.

[85] Inserter für den Zielbereich schaffen sich im Zielcontainer selbst Platz:
`copy(c.begin(), c.end(), back_inserter(target));`

**Ersetzen** von Werten bei Übereinstimmung bzw. mit zutreffendem Prädikat kann sowohl im selben Bereich (*in place*) geschehen (Bsp. S. 104) als auch durch Kopieren in einen genügend großen Zielbereich. Das Ende des kopierten Bereichs wird zurückgegeben:

```
void replace(For first, For last, T oldvalue, T newvalue);
void replace_if(For first, For last, Pred pred, T newvalue);
Out replace_copy(In first, In last, Out result, T oldvalue, T newvalue);
Out replace_copy_if(In first, In last, Out result, Pred pr, T newvalue);
```

**Entfernen** von Elementen bei Gleichheit bzw. mit zutreffendem Prädikat oder von Duplikaten in einem sortierten Bereich erfolgt mit

```
For remove(For first, For last, T value);
For remove_if(For first, For last, Pred pred);
Out remove_copy(In first, In last, Out result, T value);
Out remove_copy_if(In first, In last, Out result, Pred pred);

For unique(For first, For last);
For unique(For first, For last, Binary pred);
Out unique_copy(In first, In last, Out result);
Out unique_copy(In first, In last, Out result, Binary pred);
```

Die in-place-Version zieht nicht entfernte Objekte nach vorn. Dabei verbleiben einige alte Elemente hinter dem zurückgegebenen Ende des kompaktierten Bereichs.[86] Beim Kopieren muss der Zielbereich muss ausreichend groß sein.

**Transformationen** benutzen einen oder zwei Eingabebereiche, um mit Hilfe ein- bzw. zweistelliger Funktionen Werte in einen Zielcontainer zu schreiben. Das Ziel kann auch eine der Quellen sein. Der Ende des geschriebenen Bereichs wird zurückgegeben:

```
Out transform(In first, In last, Out result, Func func);
Out transform(In first, In last, In first2, Out result, Binary func);
```

Zweiter Eingangs- und der Zielbereich müssen groß genug sein:

```
Container summen(const Container& a, const Container& b)
{
  typedef Container::value_type T;
  Container sum;

  std::transform(begin(a), end(e),          // a[i] +
                 begin(b),                  // b[i]
                 back_inserter(sum),        // ==> sum[i]
                 [](T x, T y) { return x + y; });
  return sum;
}
```

[86] Das wirkliche Löschen muss der Aufrufer übernehmen, da der Algorithmus nichts über Container wissen kann: `c.erase(unique(c.begin(), c.end()), c.end());`

### C.15.4 Mutierende Algorithmen

Sie ändern die Elementreihenfolge, ohne Werte zu ändern, einzufügen oder wegzulassen.

**Umkehren** kann am Ort oder in einen (ausreichend großen) Zielbereich erfolgen:

```
void reverse(Bi first, Bi last);
Out reverse_copy(Bi first, Bi last, Out result);
```

**Rotieren** macht `middle` zum Anfang und hängt `[first,middle)` hinten an:

```
void rotate(For first, For middle, For last);
Out rotate_copy(For first, For middle, For last, Out result);
```

**Zufällige Umordnungen und Auswahlen** eines Bereichs entstehen mit

```
void shuffle(Ran first, Ran last, UniformRandomBitGenerator gen);
Out  sample(In first, In last, Out result, N distance,
            UniformRandomBitGenerator gen);
```

Der Aufruf `gen()` liefert Zufallszahlen aus dem Bereich `[gen.min(),gen.max()]`. Ein Generator mit gleichem Startzustand (Saatwert) liefert stets die gleiche pseudozufällige Umordnung.

**Permutieren** erzeugt, lexikographisch sortiert, bei jedem Aufruf eine Vertauschung

```
1 2 3        // 3! = 6 verschiedene Anordnungen:
1 3 2  true
2 1 3  true
2 3 1  true
3 1 2  true
3 2 1  true
1 2 3  false
```

der Elemente eines Bereichs und liefert `false`, wenn die Folge danach sortiert ist:

```
bool next_permutation(first, last);
bool next_permutation(first, last, comp); // aufsteigend
bool prev_permutation(first, last);
bool prev_permutation(first, last, comp); // absteigend
```

Nach $n!$ Permutationen ist ein sortierter Bereich mit $n$ Elementen wieder sortiert.[87] Ob zwei Bereiche durch Permutation ineinander überführt werden können, prüfen

```
bool is_permutation(For first1, For last1, For first2);
bool is_permutation(For first1, For last1, For first2, Binary pred);
```

[87] Die langsamste Art zu sortieren `while (std::next_permutation(begin(c), end(c)));` bringt auch sortierte Bereiche erst wieder durcheinander.

### C.15.5 Sortierende Algorithmen

Sie bringen Ordnung in Sequenzen. Es wird mittels `<` oder einer anderen von Nutzer angegebenen Vergleichsfunktion `comp(x,y)` aufsteigend sortiert.

**Vollständiges Sortieren** führen die Algorithmen

```
void sort(Ran first, Ran last);
void sort(Ran first, Ran last, Comp comp);
void stable_sort(Ran first, Ran last);
void stable_sort(Ran first, Ran last, Comp comp);
```

durch. Bei `stable_sort()` bleibt die Reihenfolge gleicher Elemente während des Sortiervorgangs erhalten. Dies kostet allerdings mehr Zeit: $O(n(\log n)^2)$ statt $O(n \log n)$. Ob überhaupt sortiert werden muss, kann mit linearem Zeitaufwand durch

```
bool is_sorted(For first, For last);
bool is_sorted(For first, For last, Comp comp);
For is_sorted_until(For first, For last);
For is_sorted_until(For first, For last, Comp comp);
```

entschieden werden: Der Bereich `[first,result)` ist sortiert.

**Unvollständiges Sortieren** bricht ab, wenn der Teilbereich `[first,middle)` bzw. `[result_first,result_last)` sortiert ist. Manchmal genügt es auch, wenn das $n$-te Element an der richtigen Stelle, die kleineren links und die größeren rechts davon stehen.

```
void partial_sort(Ran first, Ran middle, Ran last);
void partial_sort(Ran first, Ran middle, Ran last, Comp comp);
Ran partial_sort_copy(Ran first, Ran last,
                      Ran result_first, Ran result_last);
Ran partial_sort_copy(Ran first, Ran last,
                      Ran result_first, Ran result_last, Comp comp);
void nth_element(Ran first, Ran nth, Ran last);
void nth_element(Ran first, Ran nth, Ran last, Comp comp);
```

**Partitionieren** bringt alle Elemente mit der Eigenschaft `pred(*i)` nach vorn und liefert das Ende des „guten“ und bei Kopie auch des „schlechten“ Teilbereichs:[88]

```
Bi partition(Bi first, Bi last, Pred pred);
Bi stable_partition(Bi first, Bi last, Pred pred);
pair<Out1, Out2> partition_copy(In first, In last, Out1 good, Out2 bad,
                                Pred pred);
```

Ob ein Bereich partitioniert ist und wo die Partitionsgrenze liegt, ermitteln

```
bool is_partitioned(In first, In last, Pred pred);
For partition_point(For first, For last, Pred pred);
```

[88] Frei nach Aschenputtels Spruch `std::partition_copy(begin(erbsen), end(erbsen), std::back_inserter(toepfchen), std::back_inserter(kroepfchen), gut);`

**Mischen** fasst zwei sortierte Bereiche zu einem sortierten Bereich zusammen (Abb. 9a). Bei der in-place-Variante ist `middle` die Grenze der beiden vorsortierten Bereiche:

```
Out merge(In first, In last, In2 first2, In2 last2, Out result);
Out merge(In first, In last, In2 first2, In2 last2, Out result,
          Comp comp);
void inplace_merge(Bi first, Bi middle, Bi last);
void inplace_merge(Bi first, Bi middle, Bi last, Comp comp);
```

**Mengenoperationen** setzen ebenfalls sortierte Bereiche $M_1 =$ `[first1, last1)` und $M_2 =$ `[first2, last2)` voraus, da sie nur auf solchen effizient realisierbar sind (Abb. 9b). Der Teilmengentest $\underline{M_1} \supset \underline{M_2}$

```
bool includes(In first1, In last1, In2 first2, In2 last2);
bool includes(In first1, In last1, In2 first2, In2 last2, Comp comp);
```

prüft, ob der erste Bereich alle Elemente des zweiten enthält. Die Vereinigung $\underline{M_1} \cup \underline{M_2}$, die Schnittmenge $\underline{M_1} \cap \underline{M_2}$, die Mengendifferenz $\underline{M_1} \backslash \underline{M_2}$ und die symmetrische Differenz $\underline{M_1} \triangle \underline{M_2} = (\underline{M_1} \backslash \underline{M_2}) \cup (\underline{M_2} \backslash \underline{M_1})$

```
Out set_union(In first1, In last1, In2 first2, In2 last2, Out result);
Out set_union(In first1, In last1, In2 first2, In2 last2, Out result,
              Comp comp);
Out set_intersection(In first, In last, In2 first2, In2 last2,
                     Out result);
Out set_intersection(In first, In last, In2 first2, In2 last2,
                     Out result, Comp comp);
Out set_difference(In first, In last, In2 first2, In2 last2,
                   Out result);
Out set_difference(In first, In last, In2 first2, In2 last2,
                   Out result, Comp comp);
Out set_symmetric_difference(In first, In last, In2 first2, In2 last2,
                             Out result);
Out set_symmetric_difference(In first, In last, In2 first2, In2 last2,
                             Out result, Comp comp);
```

liefern das Ende des (ausreichend großen) Zielbereichs.

**Heap-Algorithmen** arbeiten auf einem ausgeglichenen Binärbaum, dem in der untersten Schicht nur auf der rechten Seite Knoten fehlen.[89] Zudem sind die darüberliegenden Knotenwerte niemals kleiner als ihre Kindelemente. Sind beide Forderungen erfüllt, liefert

```
bool is_heap(Ran first, Ran last);
bool is_heap(Ran first, Ran last, Comp comp);
Ran is_heap_until(Ran first, Ran last);
Ran is_heap_until(Ran first, Ran last, Comp comp);
```

[89] Wegen dieser *Formeigenschaft* ist jede Sequenz mit wahlfreiem Zugriff in einen Heap umwandelbar.

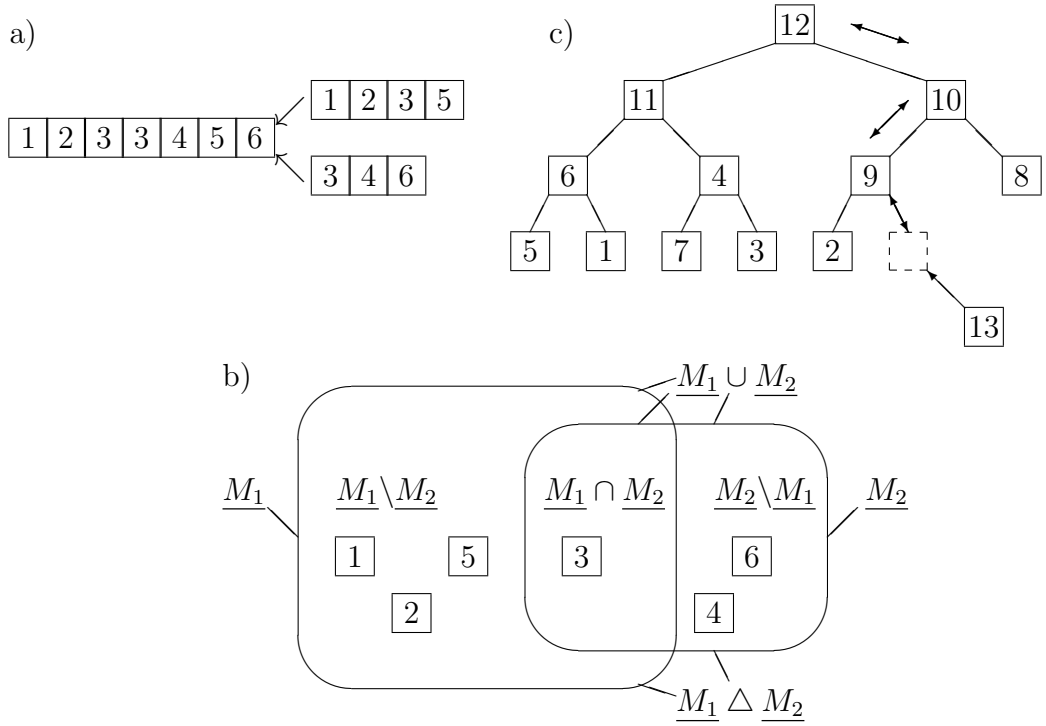



Abbildung 9: a) Mischen. b) Mengen-Operationen. c) Einfügen in Heap.

den Wert `true` bzw. das Ende des Heap-Bereichs. Die Funktionen

```
void push_heap(Ran first, Ran last);
void push_heap(Ran first, Ran last, Comp comp);
void pop_heap(Ran first, Ran last);
void pop_heap(Ran first, Ran last, Comp comp);
```

fügen das letzte Element in den davorliegenden Heap ein (Abb. 9c) bzw. stellen das Kopfelement `*first` ans Ende und ordnen die Elemente davor wieder zum Heap um. Mit

```
void make_heap(Ran first, Ran last);
void make_heap(Ran first, Ran last, Comp comp);
```

wird ein Bereich in einen Heap umgewandelt, ein Heap mittels

```
void sort_heap(Ran first, Ran last);
void sort_heap(Ran first, Ran last, Comp comp);
```

in $O(n \log n)$ wieder vollständig aufsteigend sortiert.

### C.15.6 Numerische Algorithmen

Die Funktionen `gcd(a,b)`, `lcm(a,b)` für ggT und kgV sowie `clamp(x,min,max)` zum Einschränken eines Wertes `x` in ein Intervall `[min,max]` erwartet man eher in `<cmath>`. Weitere Algorithmen aus `<numeric>` (Abb. 10) verallgemeinern mathematische Operationen auf Datenmengen.

```
void iota(For first, For last, T startvalue);
```

schreibt eine mit `startvalue` beginnende aufsteigende Folge in den angegebenen Bereich.

```
T accumulate(In first, In last, T init);
T accumulate(In first, In last, T init, Binary op);
```

kann Summen oder mit angegebener Operation `op` auch Produkte bilden oder einem anderen Zweck dienen. Der Startwert legt den Typ von Zwischen- und Endergebnis fest. Partialsummen und Nachbar-Differenzen sind Umkehroperationen dazu, sofern Operationen `op` mit den Eigenschaften von `+` und `-` benutzt werden

```
Out partial_sum(In first, In last, Out result);
Out partial_sum(In first, In last, Out result, Binary op);
Out adjacent_difference(In first, In last, Out result);
Out adjacent_difference(In first, In last, Out result, Binary op);
```

und liefern das Ende des Zielbereichs. Die Summe von Produkten oder das Skalarprodukt (mit `+` als `op` und `*` als `op2`) liefert

```
T inner_product(In first, In last, In2 first2, T init);
T inner_product(In first, In last, In2 first2, T init,
                Binary op, Binary2 op2);
```

Der zweite Bereich muss groß genug sein. Andere Operationen `op` und `op2` sind möglich.

```
#include <numeric>

void numeric_demo()
{
  int a[5], sums[5], diff[5];

  std::iota(a, a+5, 1);                    // 1,2,3,4,5
  long   summe   = std::accumulate(a, a+5, long(0)); // 15
  double produkt = std::accumulate(a, a+5, 1.0,
                                   std::multiplies<double>()); // 120
  std::partial_sum(a, a+5, sums);          // 1, 3, 6, 10, 15
  std::adjacent_difference(a, a+5, diff); // 1, 1, 1, 1, 1
  double produktsumme  = std::inner_product(a, a+5, sums, 0.0);
                         // (1*1) + (2*3) + (3*6) + (4*10) + (5*15)
  double summenprodukt = std::inner_product(a, a+5, sums, 0.0
                         std::multiplies<double>(), std::plus<int>());
                         // (1+1) * (2+3) * (3+6) * (4+10) * (5+15)
}
```

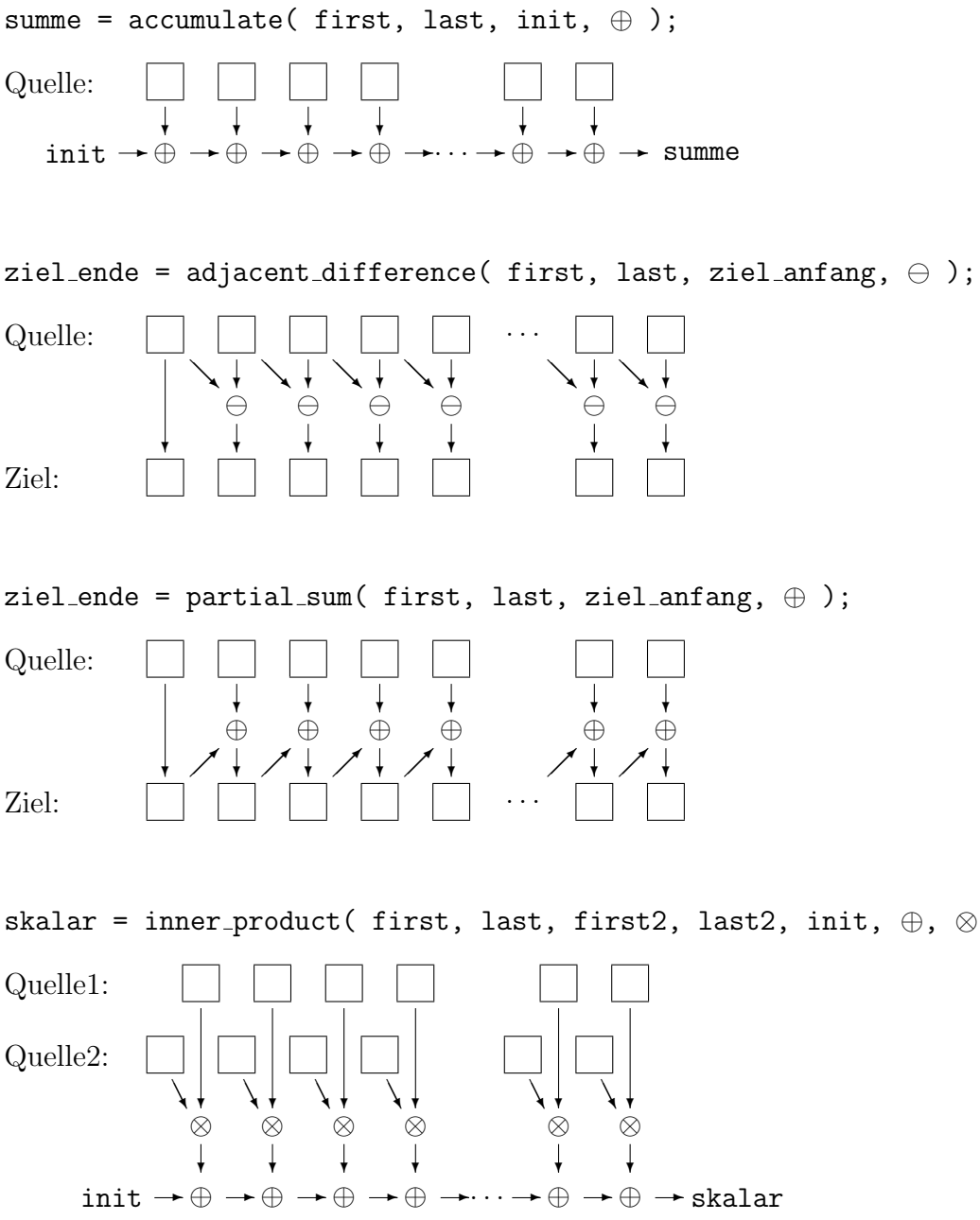



Abbildung 10: Datenflussstruktur verallgemeinerter numerischer Algorithmen. Anstelle der Operationen $\oplus, \ominus, \otimes$ können beliebige Funktionen eingesetzt werden.

## C.16 Hilfsfunktionen

### C.16.1 Funktoren

**Hilfsfunktionen** können an Algorithmen übergeben und darin aufgerufen werden. Sie arbeiten mit Werten eines Iteratorbereichs oder liefern Werte für einen Bereich:

```
int wuerfeln() { return std::rand()%6 + 1; }  // #include <cstdlib>
```

**Funktoren** (Funktionsobjekte) sind Instanzen von Klassen mit `operator()`:

```
template <class T>
struct Arithmetische_Folge {
  Arithmetische_Folge(T startwert, T schrittweite = 1)
    : wert{startwert}, inkr{schrittweite} {}
  T operator() (void) { return wert += inkr; }
private:
  T wert, inkr;
};
```

**Prädikate** sind Wahrheitsaussagen über Eigenschaften oder Beziehungen von Objekten. Ob ein Prädikat zutrifft, testen Funktoren mit `bool`-Rückgabetyp:

```
template <class T>
struct Zwischen {
  Zwischen(T a, T b) : min{a}, max{b} {}
  bool operator() (const T& x) const { return min < x && x < max; }
private:
  T min, max;
};
```

Einige der Algorithmen haben `..._if()`-Varianten, die Prädikate verwenden können:

```
#include <algorithm>
#include <vector>

void algorithmen_demo()
{
  std::vector<int> a(50), b(50);

  std::generate(begin(a), end(a), wuerfeln);
  std::generate_n(begin(b), b.size(), Arithmetische_Folge{10});
  std::replace(begin(b), end(b), 16, 61);

  int anz = std::count(begin(a), end(a), 6);
  std::replace_if(begin(a), end(a), Zwischen{0,4}, 6); // mehr Sechsen!
  anz = std::count_if(begin(b), end(b), Zwischen{18, 55});
}
```

**Lambda-Ausdrücke** als *anonyme* oder *Ad-hoc-Funktionen*[90]
`[`⟨*Capture*⟩$_{opt}$`](`⟨*Parameter*⟩`)`$_{opt}$⟨`->`*Ergebnistyp*⟩$_{opt}${⟨*Anweisungen*⟩}
definieren Funktoren zur sofortigen oder späteren Verwendung. In einfachen Fällen kann der Ergebnistyp vom Compiler ermittelt werden:

```
auto twice = [](int x) { return 2*x; }
int n = twice(3);
```

Algorithmen, die Funktoren erwarten, lassen sich flexibler formulieren. Durch Lambdas ist es seltener notwendig, Funktoren an anderer Stelle vorzudefinieren:

```
std::replace_if(begin(a), end(a),
  [](int x) { return (0 < x && x < 4; }, 6);

std::for_each(begin(a), end(a),
  [](int& x) { if (0 < x && x < 4) x = 6; });
```

Dasselbe Ergebnis erreicht man hier mit der *range-based*-`for`-Anweisung[91]

```
for(int& x : a) { if (0 < x && x < 4) x = 6; }
```

Lambda-Ausdrücke beginnen mit einer eckigen Klammer (engl. *capture list*). Darin kann angegeben werden, welche lokalen Variablen der Umgebung im Lambda-Ausdruck bekannt sein sollen. Dieses Einbeziehen der Umgebung wird *Closure* (Einschluss) genannt. Durch die Angabe `[&]` werden alle Variablen der Umgebung als Referenz übernommen, `[=]` kopiert alle Werte der Umgebung. Auch Variablenlisten und gemischte Angaben `[=, &count]` sind möglich. In Methoden definierte Lambdas können `[this]` einschließen, um an Attribute des umgebenden Objektes heranzukommen.

```
#include <algorithm>
#include <iostream>
#include <string>

void closuredemo(std::string s = "Hello, Lambda Expressions in C++0x!")
{
  char low = 'A', high = 'Z';
  int count = 0;

  std::for_each(begin(s), end(s),
    [low, high, &count] (char c)
    {
      if (low <= c && c <= high) ++count;
    }
  );
  std::cout << count << '\n';
}
```

[90] Alle Lambdas sind typverschieden, selbst wenn sie die gleiche Signatur besitzen.
[91] `for(`⟨*Typ*⟩$_{opt}$ ⟨*Element*⟩`:`⟨*Container*⟩`) ...`

Tabelle 16: Funktoren für Grundoperationen auf dem Typ `T` aus `<functional>`.

| Arithmetik | | Vergleiche | | Logik | |
|---|---|---|---|---|---|
| + | plus<T> | == | equal_to<T> | && | logical_and<T> |
| - | minus<T> | != | not_equal_to<T> | \|\| | logical_or<T> |
| - | negate<T> | < | less<T> | ! | logical_not<T> |
| * | multiplies<T> | <= | less_equal<T> | & | bit_and<T> |
| / | divides<T> | > | greater<T> | \| | bit_or<T> |
| % | modulus<T> | >= | greater_equal<T> | ^ | bit_xor<T> |

### C.16.2 Funktorenbibliothek

**Operatorobjekte** für die Grundoperationen (Tab. 16) sind wie Funktionen nutzbar:

```
#include <functional>
#include <algorithm>
#include <iterator>
#include <list>

std::list<double> funktoren(Container a, Container b, Container c)
{
  std::plus<int> Add;
  int z = Add(1, 2);   // z = 1 + 2;
                                                    // a + b ==> c
  std::transform(begin(a), end(a), begin(b), begin(c), Add);
  std::list<double> d;
  std::transform(begin(a), end(a), begin(b),        // a * b ==> d
                 std::back_inserter(d), std::multiplies<double>());
  return d;
}
```

**Polymorphe Funktionsadapter** `std::function<`⟨*Ergebnistyp(Typliste)*⟩`>` nehmen sowohl Funktoren, Funktionszeiger als auch Lambda-Ausdrücke auf. Ob ihnen ein Wert zugewiesen wurde, kann vor dem Aufruf geprüft werden:

```
#include <cmath>

void polymorphic_function_adapter()
{
  std::function<double(double)> f = (double(*)(double)) std::acos;
  f = [] (double x) { return std::acos(x); };
  f = std::negate<double>();
  if (f) std::cout << f(1) << '\n';
}
```

**Methodenzeiger** werden eingekapselt, damit die korrekte, auch virtuelle, Methode mit den richtigen Objektzeigern oder -referenzen aufgerufen wird:

```
void zeichne_und_drehe(std::list<std::shared_ptr<Figur>> x, float winkel)
{
  std::for_each(begin(x), end(x), std::mem_fn(&Figur::draw));
  std::for_each(begin(x), end(x),
    std::bind(std::mem_fn(&Figur::rotate), winkel)); // siehe unten
}
```

**Binder** sind flexible Funktionsadapter. `std::bind(`⟨*Funktor, Argumentliste*⟩`)` koppelt die Parameter eines Funktors (Funktions- oder Methodenzeigers) an Platzhalter für die Argumente des Aufrufs, an Werte, Referenzen `std::ref(`⟨*Variable*⟩`)`, konstante Referenzen `std::cref(`⟨*Variable*⟩`)` oder Ergebnisse anderer Binder-Funktoren:

```
#include <algorithm>
#include <functional>
#include <string>
#include <vector>
#include <iostream>

int bind_demo(std::vector<std::string> text)
{
  std::divides<double> div;
  double nenner = 2;

  using namespace std::placeholders;                      // _1, _2,...
  auto inverse   = std::bind(div, _2, _1);                // Platzhalter
  auto one_half  = std::bind(div, 1.0, nenner);           // Wert
  auto bruch     = std::bind(div, 1.0, std::ref(nenner)); // Referenz
  auto strlength = std::bind(&std::string::size, _1);     // Methode
  auto min4Chars = std::bind(std::greater_equal<char>(), strlength, 4);

  std::cout << inverse(1, 3) << '\n';  // div(3.0, 1.0)
  std::cout << one_half() << '\n';     // div(1.0, 2.0)
  std::cout << bruch() << '\n';        // div(1.0, nenner) == div(1, 2)
  nenner = 10;
  std::cout << bruch() << '\n';        // dto, jetzt div(1.0, 10.0)
  int n = std::count_if(begin(text), end(text), min4Chars);
  return  std::count_if(begin(text), end(text), std::not1(min4Chars));
}
```

**Negierer** `not1()` und `not2()` ermitteln das Gegenteil ein- bzw. zweistelliger Prädikate.

## C.17 Mathematik

### C.17.1 Mathematische Funktionen

Grundfunktionen aus `<cmath>` (Tab. 17) erwarten Argumente der Typen `float`, `double` oder `long double`. Das Ergebnis hat den gleichen bzw. bei verschiedenen Argumenttypen den breiteren Typ. Die Bibliothek definiert außerdem Makros (`INFINITY`, `NAN`).

### C.17.2 Komplexe Zahlen

Klassen `std::complex<T>` für komplexe Zahlen (Abb. 11) aus Gleitkommatypen `T` finden sich im Header `<complex>`. Neben Grundrechenoperationen, Ein- und Ausgabe können Winkelfunktionen, Hyperbelfunktionen, $e^z$, $\ln z$, $\log_{10} z$, Potenzen $z^{z_2}$ und Quadratwurzeln $\sqrt{z}$ (Tab. 17) berechnet werden. Die Speicheranordnung garantiert, dass Real- und Imaginärteil komplexer Zahlen mit Feldelementen gleichen Typs zur Deckung kommen:

```
std::complex<double> leben(double re = 1, double im = 2)
{
  std::complex<double> z{re, im}, i = sqrt(std::complex{-1.0});
  z = real(z) - i*imag(z); // conj(z)
  z = proj(z);             // Projektion auf Riemannkugel

  auto array = reinterpret_cast<double(&)[2]>(z);
  re = array[0]; im = array[1];

  double r   = abs(z);     // std::sqrt(norm(z))
  double phi = arg(z);     // std::atan2(imag(z), real(z))
  return std::polar(abs(z), arg(z));
}
```

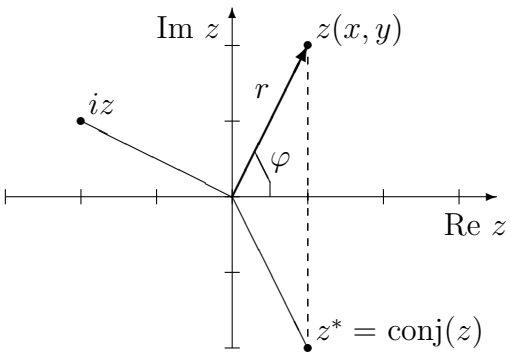



Abbildung 11: Komplexe Zahl $z$ in der Gaußschen Zahlenebene in kartesischen $(x + iy)$ und Polarkoordinaten $(r, \varphi)$, multipliziert mit $i$ und Konjugierte $z^*$.

Tabelle 17: Funktionen aus `<cmath>` im Namensraum `std` (Auswahl).

| | |
|---|---|
| | Betrag und Vorzeichen: |
| `abs(x)`, `fabs(x)` | $\|x\|$ |
| `signbit(x)` | `x < 0` |
| `copysign(x,y)` | $\|x\| * sgn(y)$ |
| | Potenzen und Wurzeln: |
| `pow(x,y)` | $x^y$ (für $x \geq 0$) |
| `sqrt(x)`, `cbrt(x)` | $\sqrt{x}$ für $x \geq 0$, $\sqrt[3]{x}$ |
| `hypot(x,y)`, `hypot(x,y,z)` | $\sqrt{x^2+y^2}$, $\sqrt{x^2+y^2+z^2}$ |
| | e-Funktion, Logarithmen: |
| `exp(x)`, `exp2(x)` | $e^x$, $2^x$ |
| `log(x)`, `log2(x)`, `log10(x)` | $\ln x$, $\log_2 x$, $\log_{10} x$ |
| `exp1m(x)`, `log1p(x)` | $e^x - 1$, $\ln(1+x)$ für $x > -1$ |
| `frexp(x, &n)` | $x \mapsto y * 2^n$ mit $\frac{1}{2} \leq y < 1$ |
| `ldexp(y, n)` | $y * 2^n$ |
| | Winkelfunktionen (im Bogenmaß): |
| `cos(x)`, `sin(x)`, `tan(x)` | $\cos x$, $\sin x$, $\tan x$ |
| `acos(x)`, `asin(x)`, | $\arccos x$, $\arcsin x$ für $-1 \leq x \leq 1$ |
| `atan(x)`, `atan2(y,x)` | $\arctan x$, $\arctan \frac{y}{x}$ (bei $x = 0 \Rightarrow y \neq 0$) |
| | Hyperbelfunktionen: |
| `cosh(x)`, `sinh(x)`, `tanh(x)` | $\cosh x$, $\sinh x$, $\tanh x$ |
| `acosh(x)`, `asinh(x)`, `atanh(x)` | $\operatorname{arcosh} x$, $\operatorname{arsinh} x$, $\operatorname{artanh} x$ |
| | Gamma- und Gaußsche Fehlerfunktion: |
| `lgamma(x)`, `tgamma(x)` | $\ln \Gamma(x)$, $\Gamma(x) = (x-1)!$ für $x > 0$ |
| `erf(x)`, `erfc(x)` | $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$, `1-erf(x)` $= \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$ |
| | nächste ganze Gleitkommazahl: |
| `floor(x)`, `ceil(x)`, `trunc(x)`, `round(x)` | $\lfloor x \rfloor$, $\lceil x \rceil$, $[x]$, ab $n.5$ weg von 0 |
| `rint(x)`, `nearbyint(x)` | in aktuelle Rundungsrichtung |
| | mit/ohne Setzen der inexact-Ausnahme |
| `lround(x)`, `llround(x)` | Runden zu Ganzzahl `long`, `long long` |
| | Divisionsrest $r = x - n * y$ |
| `fmod(x,y)` | mit $\|r\| < \|y\|$, $sgn(r) = sgn(y)$ |
| `remainder(x,y)`, `remquo(x,y, &n)` | nächstes n, gerade bei $\|n - x/y\| = 0.5$ |
| | Gleitkommadarstellung: |
| `nextafter(x,y)` | $x \pm \varepsilon$ in Richtung y |
| `nexttoward(x,y)` | dto. mit `long double y` |
| `isfinite(x)`, `isnormal(x)`, `isinf(x)` | endlich, normalisiert, $x = \pm\infty$ |
| `isunordered(x,y)` | `isnan(x)` oder `isnan(y)` |
| `fmin(x,y)`, `fmax(x,y)` | kleinster/grösster Wert, auch mit NANs |
| `fdim(x,y)` | $x - y$ für $x > y$, sonst 0 |
| `fma(x,y,z)` | `x*y+z` mit nur einer Rundung |

### C.17.3 Tupel

$n$-Tupel $(a_1, a_2, ..., a_n)$ sind heterogene Gruppierungen von Werten aus $n$ Grundmengen. Die Schablone `template <class...Types> class tuple` nimmt eine beliebige, feste Anzahl von Argumenten verschiedener Typen auf, `std::pair<X,Y>` nur zwei. Die Funktionen `std::make_tuple(`⟨*Wertliste*⟩`)` zum Verpacken, `std::tie(`⟨*Referenzliste*⟩`)` und `std::get<`⟨*Elementnummer*⟩`>(`⟨*Tupel*⟩`)` zum Entpacken erleichtern die Nutzung:

```
#include <functional> // ref
#include <tuple>
#include <utility>    // pair

void tupledemo(char c = 'A', int i = 200, double d = 3.14)
{
  auto t = std::tuple{c, i, d};
  c = std::get<0>(t);
  i = std::get<1>(t);
  d = std::get<2>(t);
  std::tie(c, i, d) = t;                          // statt so:
  std::make_tuple(std::ref(c), std::ref(i), std::ref(d)) = t;
  std::tie(c, std::ignore, d) = t;

  auto p = std::pair{c, std::pair{i,d}};
  c = p.first;
  std::tie(i, d) = p.second;
}
```

Der Platzhalter `std::ignore` nimmt uninteressante Bestandteile beim Entpacken auf. Gäbe es Tupel nicht, wären sie rekursiv über geschachtelte 2-Tupel `std::pair<X,Y>` simulierbar.[92] Arrays, Tupel, Paare und andere Strukturen lassen sich ab C++17 auch mit „structured bindings“ entpacken:

```
double spread(const Container& c)
{
  auto [min, max] = std::minmax_element(begin(c), end(c));
  return max != min ? *max - *min : 0;
}
```

### C.17.4 Verhältnisse

Mit `std::ratio<`⟨*Zähler*⟩`,`⟨*Nenner*⟩`>` lassen sich zur Übersetzungszeit berechnete Brüche aus Ganzzahlen wie die SI-Vorsätze (Tab. 18) definieren. Ihre Bestandteile `nom` und `den` müssen in `intmax_t` passen.[93] Sie finden u.a. Anwendung in der Bibliothek `<chrono>`.

►C.18

[92] Lisp-Programmierern ist dieser Gedanke vertraut, aus (1,2,3) würde (1,(2,3)). Allerdings lassen sich nur `std::tuple<X,Y>` und `std::pair<X,Y>` direkt ineinander überführen.

[93] Daher sind `zeta`, `yotta`, `zepto` und `yocto` nicht auf allen Systemen definiert.

Tabelle 18: SI-Vorsätze aus `<ratio>` im Namensraum `std`.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `deca` | $10:1$ | `deci` | $1:10$ | `tera` | $10^{12}:1$ | `pico` | $1:10^{12}$ |
| `hecto` | $100:1$ | `centi` | $1:100$ | `peta` | $10^{15}:1$ | `femto` | $1:10^{15}$ |
| `kilo` | $1000:1$ | `milli` | $1:1000$ | `exa` | $10^{18}:1$ | `atto` | $1:10^{18}$ |
| `mega` | $10^6:1$ | `micro` | $1:10^6$ | `zetta` | $10^{21}:1$ | `zepto` | $1:10^{21}$ |
| `giga` | $10^9:1$ | `nano` | $1:10^9$ | `yotta` | $10^{24}:1$ | `yocto` | $1:10^{24}$ |

### C.17.5 Zufallszahlen

**Überblick** Die Funktion `std::rand()` erzeugt pseudozufällige Zahlen:[94]

```
#include <cstdlib>
#include <ctime>
#include <iostream>

void zufall0()
{
  unsigned long seed = (unsigned long) std::time(0);
  std::srand(seed);    // nur 1x im Programm!
  for (int i = 0; i < 20; ++i)
    std::cout << std::rand() << ' ';
}
```

C++ stellt zudem miteinander kombinierbare Generatoren (*Engines*), Saatwerte[95] und Verteilungsfunktionen bereit, die höheren Ansprüchen genügen:

```
#include <random>
#include <functional>

void zufall1()
{
  std::random_device rd;
  std::mt19937 engine{rd()};
  // std::mt19937 engine{(unsigned long)std::time(0)};
  std::normal_distribution<> normal;
  std::function<double()> rnd = std::bind(normal, engine);

  for (int i = 0; i < 20; ++i)
    std::cout << rnd() << ' ';
}
```

[94] Es sind keine echt zufälligen Zahlen, da sie mittels eines Algorithmus aus einem Saatwert `srand(seed)` erzeugt werden. `rand()` ist für nebenläufige Programme ungeeignet. Zudem ist die erzeugte Zufallsfolge häufig von minderer Qualität.

[95] `std::random_device` ist bei MinGW g++ 7.2 unter Windows (noch?) nicht zufällig. Für kryptographisch sichere Saatwerte muss auf Betriebssystemmittel zurückgegriffen werden.

Tabelle 19: Zufallszahlgeneratoren aus `<random>` im Namensraum `std`.

| | |
|---|---|
| `minstd_rand0` | LCG mit $a = 16807, c = 0, m = 2^{31} - 1$ |
| `minstd_rand` | LCG mit $a = 48271, c = 0, m = 2^{31} - 1$ |
| `mt19937` | Mersenne Twister 32bit |
| `mt19937_64` | Mersenne Twister 64bit |
| `ranlux24_base` | subtraktiver Generator mit Übertrag, 24bit |
| `ranlux48_base` | dto, 48bit |
| `ranlux24` | nimmt 23 von 223 erzeugten Werten |
| `ranlux48` | nimmt 11 von 389 erzeugten Werten |
| `knuth_b` | Shuffle-Algorithmus von Donald E. Knuth |
| `default_random_engine` | einfacher Generator, implementierungsabhängig |

**Generatoren** sind Funktionsobjekte, die beim Aufruf `g()` einen Ganzzahlwert aus dem geschlossenen Intervall [`g.min()`, `g.max()`] liefern. Ein Funktionsobjekt `g` heißt *Engine*, wenn es im Konstruktor eine Saatsequenz übernehmen kann, die auch mit `g.seed(seq)` neu wiederherstellbar ist. Mit `g.discard(n)` wird eine Anzahl von Werten übersprungen. Der Zustand einer Engine lässt sich in einen Strom schreiben bzw. aus diesem wiederherstellen und mit Vergleichsoperatoren die Gleichheit der Zustände prüfen:

```
std::mt19937 engine1, engine2;
std::stringstream input;
input << engine1;
input >> engine2;
assert(engine1 == engine2);
```

Vorgefertigte Generator-Engines (Tab. 19) sind Spezialisierungen von Schablonen. Lineare Kongruenzgeneratoren (LCG) erzeugen Werte $x_{n+1} = (ax_n + c) \mod m$, subtraktive mit $x_{n+1} = x_{n-s} - x_{n-r} - c \mod m$. Mersenne-Twister bieten sehr lange Perioden. Daneben gibt es eine Reihe von Adaptern. Eine `discard_block_engine<Engine, p, r>` überspringt nach $r$ Werten die restlichen Werte einer von `engine` erzeugten Folge aus $p$ Werten. Eine `independent_bits_engine<Engine, w, zieltyp>` kombiniert $w$ Bits der Basis-Engine zum Zieltyp. Die `shuffle_order_engine<Engine, k>` liefert jeweils $k$ Werte der Basis-Engine in veränderter Reihenfolge.

**Verteilungen** `D(`⟨*parameter*⟩`)` erzeugen mit dem Generator `g` beim Aufruf `d(g)` bzw. `d(g, d.param())` eine Zufallszahl nach dem zugehörigen Verteilungsgesetz (Tab. 20) im Intervall [`d.min()`, `d.max()`]. Die Parameter der Verteilung lassen sich auslesen und mit `d.param(p)` neu setzen sowie mit I/O-Operatoren in Datenströme schreiben bzw. daraus lesen. Nach `d.reset()` hängen nachfolgende Werte nicht mehr von bisher erzeugten ab.

**Saatfolgen** `S(first, last)` wie von `std::random_device` geliefert initialisieren Engines, `s.generate(first, last)` befüllt den angegebenen Bereich, `s.param(out)` schreibt `s.size()` Werte in den Ausgabeiterator `out`.

Tabelle 20: Zufallsverteilungen aus `<random>` im Namensraum `std`.

| | | |
|---|---|---|
| `uniform_int_distribution(a = 0, b=numeric_limits<int>::max())` | $P(i\|a,b) = 1/(b-a+1)$ | $a \le i \le b$ |
| `uniform_real_distribution (a = 0.0, b = 1.0)` | $p(x\|a,b) = 1/(b-a)$ | $a \le x < b$ |
| `bernoulli_distribution(p = 0.5)` | $P(b\|p) = \begin{cases} p & \text{für } b = \\ 1-p & \end{cases}$ | $\begin{cases} \texttt{true} \\ \texttt{false} \end{cases}$ |
| `binomial_distribution (n=1, p=0.5)` | $P(i\|n,p) = \binom{n}{i} p^i (1-p)^{n-i}$ | $0 \le i \le n$ |
| `geometric_distribution(p = 0.5)` | $P(i\|p) = p(1-p)^i$ | $0 \le i$ |
| `negative_binomial_distribution (n = 1, p = 0.5)` | $P(i\|n,p) = \binom{n+i+1}{i} p^n (1-p)^i$ | $0 \le i$ |
| `poisson_distribution(mu = 1.0)` | $P(i\|\mu) = \frac{e^{-\mu}\mu^i}{i!}$ | $0 \le i$ |
| `exponential_distribution (lambda = 1.0)` | $p(x\|\lambda) = \lambda e^{-\lambda x}$ | $0 < x$ |
| `gamma_distribution (alpha = 1.0, beta = 1.0)` | $p(x\|\alpha,\beta) = \frac{e^{-x/\beta}}{\beta^\alpha \Gamma(\alpha)} x^{\alpha-1}$ | $0 < x$ |
| `weibull_distribution (a = 1.0, b = 1.0)` | $p(x\|a,b) = \frac{a}{b}\left(\frac{x}{b}\right)^{a-1} e^{-\left(\frac{x}{b}\right)^a}$ | $0 \le x$ |
| `extreme_value_distribution (a = 0.0, b = 1.0)` | $p(x\|a,b) = \frac{1}{b} e^{-\gamma - e^{-\gamma}}$, $\gamma = \frac{x-a}{b}$ | $x \in \mathbb{R}$ |
| `normal_distribution (mu = 0.0, sigma = 1.0)` | $p(x\|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ | $x \in \mathbb{R}$ |
| `lognormal_distribution (mu = 0.0, sigma = 1.0)` | $p(x\|\mu,\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}$ | $0 < x$ |
| `chi_squared_distribution (n = 1)` | $p(x\|n) = \frac{x^{(n/2)-1} e^{-x/2}}{\Gamma(n/2) 2^{n/2}}$ | $0 < x$ |
| `cauchy_distribution (a = 0.0, b = 1.0)` | $p(x\|a,b) = \frac{1}{\pi} \frac{b}{(x-a)^2+b^2}$ | $x \in \mathbb{R}$ |
| `fisher_f_distribution (m = 1, n = 1)` | $p(x\|m,n) = \frac{\Gamma((m+n)/2))}{\Gamma(m/2)\Gamma(n/2)} \left(\frac{m}{n}\right)^{m/2} \cdot x^{m/2-1} \left(1 + \frac{mx}{n}\right)^{-(m+n)/2}$ | $0 \le x$ |
| `student_t_distribution (n = 1)` | $p(x\|n) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \cdot \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}$ | $x \in \mathbb{R}$ |
| `discrete_distribution (firstW, lastW)` | $P(i\|p_0,...,p_{n-1}) = \frac{w_i}{\sum_{k=0}^{n-1} w_k}$ | $0 \le i < n$ $w_i > 0$ |
| `piecewise_constant_distribution (firstb, lastb, firstW)` | $p(x\|b_0,...,b_n,\rho_0,...,\rho_{n-1}) = \frac{w_i}{(b_{i+1}-b_i)\sum_{k=0}^{n-1} w_k}$, | |
| `piece_linear_distribution (firstb, lastb, firstW)` | $p(x\|b_0,...,b_n,\rho_0,...,\rho_n) = \rho_i \frac{b_{i+1}-x}{b_{i+1}-b_i} + \rho_{i+1} \frac{x-b_i}{b_{i+1}-b_i}$, $\rho_i = \frac{w_i}{\frac{1}{2}\sum_{k=0}^{n-1}(w_k+w_{k-1})(b_{k+1}-b_k)}$ | $b_i \le x < b_{i+1}$ |

### C.17.6 Bitmengen

Gegenüber `std::array<bool,N>` packt `std::bitset<N>` eine feste Anzahl von Bits in eine kompakte, speichereffiziente Form. Zur Initialisierung sind Ganzzahlen, Zeichenketten und Sequenzen boolescher Werte geeignet. Bitweise und Bitschiebeoperatoren sowie Vergleichs- und Ein-/Ausgabe-Operationen sind nutzbar:

```
#include <bitset>

auto answer()
{
  std::bitset<32> b{"1100000010101000"}, netmask{0xFFFFFF00};
  b <<= 16;
  b |= 42;
  netmask.flip();
  return (b & netmask);
}
```

Der Elementzugriff auf ein einzelnes Bit erfolgt mit `b[i]`. Methoden `b.set(i,wert)` zum Setzen, `b.reset(i)` zum Löschen und `b.flip(i)` zum Umkehren einzelner Bits bzw. ohne Index `b.set(wert)`, `b.reset()` und `b.flip()` für alle Bits sind definiert. Abfragemethoden `b.size()`, `b.count()`, `b.all()`, `b.any()` und `b.none()` ermitteln Anzahl aller Bits bzw. testen, ob und wieviele Bits gesetzt sind.

Mit `b.to_ulong()` und `b.to_ullong()` wird der Ganzzahlwert der Bitfolge geliefert. Kann dieser nicht im Zieltyp untergebracht werden, wird ein `std::overflow_error` geworfen. Die Methode `b.to_string()` erzeugt eine Zeichenkette aus Nullen und Einsen[96], dabei steht das niederwertigste Bit ganz rechts.

### C.17.7 Wertfelder

**Elementweise Operationen** und Funktionsberechnungen ohne die explizite Formulierung von Schleifen bei optimalem Zeitverhalten waren das Ziel der Entwicklung von `std::valarray<T>`.[97] Erlaubt sind die bei den numerischen Grundtypen `T` üblichen Operationen. Sind beiden Operanden Wertfelder, müssen sie gleich viele Elemente haben.[98]

Neben dem Elementzugriff `v[i]` mit `i < v.size()` gibt es nur wenige Methoden: `v.min()`, `v.max()`, `v.sum()` sowie `v.cshift(n)` und `v.shift(n)` zum (zirkulären) Links- bzw. Rechtsschieben, das herausgeschobene Elemente bzw. Nullen am anderen Ende einfügt. Mit `v.apply(f)` lassen sich Funktionen der Signatur `T f(T)` auf jedes Element anwenden. Für die Funktionen aus `<cmath>` existieren Varianten, die auf `valarray<T>` elementweise arbeiten:

[96] Statt `toString('0','1')` können andere Zeichen für `true` und `false` vereinbart werden.

[97] Die Array-Operationen aus Fortran 90 lieferten die Idee zu dieser Bibliothek. Sie führt ein Schattendasein bei Hochleistungsberechnungen. Einige Compileranbieter haben für Multiprozessorsysteme optimierte Implementierungen.

[98] Andernfalls ist das Verhalten nicht definiert. Eine Überprüfung findet weder beim Übersetzen noch beim Ausführen statt.

```
#include <valarray>

void elementweise_operation()
{
  double arr[] = { 1.1, 2.2, 3.3, 4.4 };
  std::valarray<double> v1(4);
  std::valarray<double> v2(arr, 4);
  std::valarray<double> v3(3.1, 4); // andersherum bei vector<T>!
  std::valarray<double> v4({0, 1, 4, 9});

  v1 = 2.0 * (v3-v2) + sqrt(v4); // 4 2.8 1.6 0.4
  v1.cshift(1);                  // 2.8 1.6 0.4 4
}
```

**Auswahl von Arrayelementen** durch Indexfelder, Bitmasken, Vergleichsoperationen und (verallgemeinerte) Scheiben (`slice` bzw. `gslice`) erzeugt Hilfsobjekte vom Typ `indirect_array<T>`, `mask_array<T>`, `slice_array<T>` bzw. `gslice_array<T>`. Diese halten Referenzen auf die ausgewählten Elemente, die Werte oder Wertfelder elementweise übernehmen können und sich in Wertfelder kopieren lassen. Zur Veranschaulichung der Auswahl ein Beispiel mit einem Wertfeld aus Buchstaben:

```
void selektieren()
{
  std::valarray<char> v("abcdefghijklmnopqrstuvwxyz", 26);

  std::valarray<size_t> index = { 7,6,5,4,3,2,1 };
  std::valarray<bool>   mask1 = {false, false, true, false, true, true};
  std::valarray<bool>   mask2 = v < 'm';
  std::slice  s(4, 15, 2);            // start, anzahl, schrittweite
  std::gslice g(4, {3, 4}, {7, 2});   // start, anzahlen, schrittweiten

  std::valarray<char> vi  = v[index]; // hgfedcb
  std::valarray<char> vm1 = v[mask1]; // cef
  std::valarray<char> vm2 = v[mask2]; // abcdefghijkl
  std::valarray<char> vs  = v[s];     // egikmoqsuwy
  std::valarray<char> vg  = v[g];     // egiklnprsuwy

  v[mask2] = '_';       // ___________mnopqrstuvwxyz
}
```

## C.18 Zeit

### C.18.1 Abstraktion vom Rechnertakt

Die Bibliothek `<chrono>` definiert Typen für Zeitspannen, Zeitpunkte und Uhren im Namensraum `std::chrono` unabhängig von den im Lauf der Jahrzehnte immer kürzer gewordenen Rechnertakten.

**Zeitspannen** `duration<`⟨*Tickzahl*⟩`,` ⟨*Zeiteinheit*⟩`>` werden als Zahlen (Ticks) einer Zeiteinheit[99] dargestellt. Neue Zeitspannetypen können geschaffen werden:

```
#include <chrono>
#include <iostream>

void es_dauert()
{
  using namespace std::chrono;
  seconds day = hours{23} + minutes{56} + seconds{4};
  milliseconds ms = day;
  duration<double, std::ratio<1,24>> pics = day;

  std::cout << day.count()  << " sec = "
            << ms.count()   << " ms = "
            << pics.count() << " Filmbilder\n";
}
```

Rechenoperationen (Grundrechenarten, Vergleiche) erfolgen weitgehend beim Übersetzen. Die Umrechnung in kleinere Zeitspanne-Einheiten ist immer möglich, umgekehrt jedoch nur in Gleitkommatypen oder durch Cast:

```
  seconds s = ms;      // Fehler: Genauigkeitsverlust
  minutes m = duration_cast<minutes>(ms); // abrunden
```

**Uhren** messen Zeitspannen ausgehend von ihrem willkürlich festgelegten Anfangszeitpunkt, Epoche genannt.[100] Sie sind dünne Wrapperklassen um die Betriebssystemmittel zur Zeitmessung. Jede Uhr besitzt einen Zahltyp `rep`, eine Verhältniszahltyp `period` einen Zeitdauer-Typ `duration`, einen Typ für Zeitpunkte `time_point` und eine statische Klassenfunktion `now()`, die den aktuellen Zeitpunkt liefert.

Die Uhren `system_clock` und `high_resolution_clock` sind vordefiniert. Bei der Uhr `steady_clock` wird zudem garantiert, dass die von ihr nacheinander gelieferten Zeitpunkte niemals „rückwärts“ laufen. Die Auflösegenauigkeit einer Uhr kann weniger als eine Nanosekunde betragen, daher wird die Zeitspanne als Gleitkommazahl ermittelt:[101]

[99] Die Typen `hours`, `minutes`, `seconds`, `milliseconds`, `nanoseconds`. sind als Ganzzahlverhältnisse `std::ratio<`⟨*z*⟩,⟨*n*⟩`>` von Sekunden definiert.

[100] Das kann der 1. Januar 1970 0:00 GMT oder auch der Start des laufenden Prozesses sein.

[101] Siehe Howard Hinnant: How to get the precision of `high_resolution_clock`, Stack Overflow, http://stackoverflow.com/a/8387129/831725.

```
template <class Clock>
double precision_in_nanoseconds()
{
  Clock::duration tick{1};
  std::chrono::duration<double, std::nano> ns = tick;
  return ns.count();
}
```

**Zeitpunkte** `time_point<`⟨*Uhr*⟩`,`⟨*Zeitspanne*⟩`>` sind auf ihre Uhr bezogen:

```
template <class Clock>
Clock::rep ticks()
{
  typename Clock::time_point epoch, now = Clock::now();
  typename Clock::duration gone = now.time_since_epoch();
  return gone.count(); // Ticks seit Epoche
}
```

Die Differenz zweier Zeitpunkte ist eine Zeitspanne `d = p2-p1`. Zu einem Zeitpunkt lässt sich eine Zeitspanne addieren oder subtrahieren und liefert einen Zeitpunkt `p2 = p1+d`. Vergleiche `p1 < p2` erlauben Formulierungen wie davor und danach.

### C.18.2 Rückgriff auf C-Bibliothek

**Umwandlung in Datum und Anzeige** sind in `<chrono>` nicht festgelegt. Dafür existieren Mittel der Bibliothek `<ctime>` und I/O-Manipulatoren:

```
#include <iomanip> // put_time()

void vorbei()
{
  using namespace std::chrono;
  system_clock::time_point gestern = system_clock::now() - hours{24};
  std::time_t t = system_clock::to_time_t(gestern);
  std::cout << "gestern zur selben Zeit : "
            << std::put_time(std::localtime(&t), "%F %T") << '\n';
}
```

Diese enthält einen Ganzzahltyp `time_t`[102], eine Struktur `tm` mit Komponenten für Tag, Monat, Jahr (seit 1900), Stunde, Minute und Sekunde sowie Funktionen `mktime()`, `gmtime()` und `localtime()` zur Umwandlung zwischen diesen.

**Uhrticks seit Programmstart** liefert `std::clock()`, in `CLOCKS_PER_SEC` ist die Anzahl je Sekunde hinterlegt. Die Zeitmessung mit `std::time(&t)` erfolgt dagegen nur sekundengenau. Beide Varianten der Zeitmessung werden durch die `<chrono>`-Bibliothek überflüssig.

[102] Sekunden seit 1970, wird wohl im Jahr 2038 überlaufen...

## C.19 Datenkapseln

Datenkapseln für einen nur eventuell vorhandenen Wert eines festgelegten Typs, für eine vorbestimmte Menge von Typen oder beliebigen Typs werden in C++17 eingeführt. Sie übernehmen die Speicherplatzverwaltung und die Kontrolle über die Information, ob ein Wert verfügbar ist. Einen Überblick zu den Schnittstellen gibt Tabelle 21.

### C.19.1 `std::optional<T>`

Manchmal ist kein sinnvoller Rückgabewert[103] angebbar:

```
std::optional<int> find_smallest(std::vector<int> v)
{
  if (v.empty()) return {};
  return *min_element(begin(v), end(v));
}
```

Auch in Datenstrukturen finden sich Anwendungen:

```
struct Person
{
  std::string name;
  std::optional<Address> address;
};

Person p { "me" };
auto& o = p.address;
```

**Vorhandensein eines Wertes** ist mit `if (o) ...` oder `o.has_value()` prüfbar.

**Zugriff** auf den vorhandenen Wert bieten `*o` und `o.value()`. Bei nicht vorhandenem Wert kann ein `o.valueOr(`⟨*ersatzwert*⟩`)` angeben werden.

**Löschen** des Wertes erfolgt mit `o.reset()` oder durch Zuweisung von `std::null_opt`.

### C.19.2 `std::any`

Diese Kapsel ist als Ersatz für `void*` gedacht. Die Typinformation wird intern mitgeführt. Der Nutzer sollte wissen, welchen Typ der gekapselte Wert haben kann. `any_cast(a)` zum Entpacken wirft bei Nichtübereinstimmung des Typs eine Ausnahme, `any_cast(&a)` liefert einen Nullzeiger (siehe Tabelle 21).

[103] Zeiger, „magische“ Werte wie `-1` oder `bool`-Rückgabewerte in Verbindung mit Referenzvariablen waren ein schlechter Ersatz für diesen Anwendungsfall.

Tabelle 21: Datenkapseln `<any>` `<optional>` `<variant>`.
Erschaffen und Wert zuweisen, auf Typ und Wert testen, Wert entpacken und löschen, mögliche Ausnahmen.

| `any a`<br>`make_any<T>(args)` | `optional<T> o`<br>`make_optional<T>(args)` | `variant<Ts...> v`<br>`variant<monostate,...>` |
|---|---|---|
| `a = 42`<br>`a.emplace<T>(args)` | `o = 42`<br>`o.emplace(args)` | `v = 42`<br>`v.emplace<T>(args)`<br>`v.emplace<Index>(args)` |
| `a.has_value()` | `o.has_value()`<br>`if(o)` | |
| `a.type()`<br>`== typeid(T)` | | `v.index()`<br>`holds_alternative<T>(v)` |
| `any_cast<T>(a)`†<br>`any_cast<T>(&a)`<br>`!= nullptr` | `o.value()`†<br>`*o`[a]<br>`o.valueOr(42)` | `get<T>(v)`†<br>`get<Index>(v)`†<br>`visit(visitor, v)` |
| `a.reset()` | `o.reset()`<br>`o = null_opt` | |
| †`bad_any_cast`<br>`:bad_cast` | †`bad_optional_access`<br>`:exception` | †`bad_variant_access`<br>`:exception` |

[a] Undefiniertes Verhalten, wenn kein Wert vorhanden ist.

### C.19.3 `std::variant<T1,...,Tn>`

Variante Typen sind ein typsicherer Ersatz für `union`: Werte nicht verwandter Typen nutzen einen gemeinsamen Speicherplatz. Die Kapsel kümmert sich um Speicherplatz, Typprüfung und Lebensdauer: Konstruktoren und Destruktoren werden aufgerufen.

```
struct cpp11 {};
struct cpp14 {};
struct cpp17 {};
struct cppXY { std::string name; };
using Variant = std::variant<std::monostate, cpp11, cpp14, cpp17, cppXY>;
```

**Erschaffen** eines `variant`-Objekts bildet ein Objekt des zuerst aufgeführten Typs mit dessen Standardkonstruktor. Dafür kann `std::monostate` eingesetzt werden.

**Ändern** des gekapselten Wertes geschieht durch Zuweisung. `emplace<`⟨*Typ*⟩`>(`⟨*args*⟩`)` kann Argumente für den Konstruktor des erschaffenden Objektes übernehmen.

```
void zuweisen(Variant& v)
{
  v.emplace<cpp17>();
  v = cppXY{"C++20"};
}
```

**Typabfragen und Entpacken** `v.index()` bzw. `holds_alternative<`⟨*Typ*⟩`>(v)` und `get<`⟨*Index*⟩`>(v)` bzw. `get<`⟨*Typ*⟩`>(v)` sind möglich. Leeren lässt sich die Kapsel nicht — `std::monostate` kann, wenn notwendig, für eine leere Kapsel stehen.

```
void loeschen(Variant& v)
{
  std::cout << v.index() << '\n';
  v = std::monostate{};
  std::cout << v.index() << '\n';
}
```

**Besucher** sind Funktionsobjekte, die für jeden möglichen enthaltenen Typ geeignete Operationen aufrufen. Die Ergebnisse müssen einen einheitlichen Typ haben, damit der Ergebnistyp von `std::visit(`⟨*Besucher*⟩`,`⟨*Variant*⟩`)` hergeleitet werden kann.

```
struct Besucher
{
  void operator()(std::monostate) { std::cout <<  "unbekannt\n"; }
  void operator()(cpp11) { std::cout <<  "C++11\n"; }
  void operator()(cpp14) { std::cout <<  "C++14\n"; }
  void operator()(cpp17) { std::cout <<  "C++17\n"; }
  void operator()(cppXY x) { std::cout <<  x.name << "\n"; }
};

void besuchen(const Variant& v)
{
  std::visit(Besucher{}, v);
}
```

Unangemeldete Besucher mit Lambda-Ausdrücken lassen mit einem Trick[104] bilden:

```
template<class... Ts> struct overloaded:Ts...{ using Ts::operator()...;};
template<class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

void besuchen2(const Variant& v)
{
  std::visit(overloaded{
    [](std::monostate) { std::cout <<  "unbekannt\n"; },
    [](cpp11) { std::cout <<  "C++11\n"; },
    [](cpp14) { std::cout <<  "C++14\n"; },
    [](cpp17) { std::cout <<  "C++17\n"; },
    [](cppXY x) { std::cout <<  x.name << "\n"; },
  }, v);
}
```

[104] Quelle: https://en.cppreference.com/w/cpp/utility/variant/visit — Nichts für Anfänger: Variadische Templates, Faltausdrücke und Template argument deduction guides in Kombination. Ein Vorschlag für die Standardbibliothek https://wg21.link/P0051 wird vielleicht nach C++20 realisiert.

## C.20 Dateisystem

Für die Bibliothek wird ein eigener Namensraum genutzt.[105]

```
#if __has_include(<filesystem>)
  #include <filesystem>
  namespace fs = std::filesystem;
#elif __has_include(<experimental/filesystem>)
  #include <experimental/filesystem>
  namespace fs = std::experimental::filesystem;
  // g++ : link with -lstdc++fs
#elif __has_include(<boost/filesystem.hpp>)
  #include <boost/filesystem.hpp>
  namespace fs = boost::filesystem;
  // g++ link with -lboost_system -lboost_filesystem
#else
  #error No filesystem library.
#endif
```

### C.20.1 Pfade

```
      Wurzelverzeichnis
      v              Dateiname
    E:/temp/a/simple/example.txt
    /  ~~~~~~~~~~~~~ _______ ~~~
    LW  Relativpfad   Stamm  Erweiterung
```

**Dateipfade** `fs::path` beschreiben den Weg zu einer *Datei* im Dateisystem absolut zur Wurzel oder relativ zu `fs::current_path()`. Pfade bestehen aus einem optionalen `p.root_name()`[106], optionalem `p.root_directory()` und weiteren Teilen, die bis auf den letzten Namen von Verzeichnissen sein müssen. *Verzeichnisse* sind ebenfalls Dateien, auch `.` und `..` (aktuelles/übergeordnetes Verzeichnis). Dateinamen `p.filename()` bestehen oft aus zwei durch Punkt getrennten Teilen, `p.stem()` und `p.extension()`. Über die Bestandteile eines Pfades kann iteriert werden, sie werden als *quoted strings* ausgegeben:

```
for (const auto& e : fs::path("E:/temp/a/simple/example.txt"))
  std::cout << e << ' ';
// "E:" "/" "temp" "a" "simple" "example.txt"
```

`fs::exists(p)` prüft, ob ein Pfad einer Datei im Dateisystem entspricht.

Der Quellcode auf der Folgeseite liefert eine Ausgabe ähnlich dem UNIX-Befehl `ls`. Die Funktionen und Methoden werfen im Fehlerfall eine `fs::filesystem_error`-Ausnahme. Zu vielen gibt es Varianten mit einem zusätzlichen `std::error_code`-Parameter, der keine Ausnahme wirft.

[105] Vor der Integration in C++17 wurde zunächst Filesystem TS im Namensraum `std::experimental` veröffentlicht, notfalls kann diese Version oder Boost.Filesystem genutzt werden.

[106] Unter Windows ein Laufwerk `C:` oder `//server`. Pfadtrenner `fs::path::preferred_separator` sind je nach Betriebsystem `/` oder \. Im Quelltext ist `/` einfacher zu schreiben.

**Berechtigungen** regeln den Zugriff für Eigentümer und andere Nutzer.[107]

```
auto perm_string(fs::perms p)
{
  auto table = { std::pair {fs::perms::owner_read,  "-r"},
                           {fs::perms::owner_write, "-w"},
                           {fs::perms::owner_exec,  "-x"},
                           {fs::perms::group_read,  "-r"},
                           {fs::perms::group_write, "-w"},
                           {fs::perms::group_exec,  "-x"},
                           {fs::perms::others_read, "-r"},
                           {fs::perms::others_write,"-w"},
                           {fs::perms::others_exec, "-x"} };
  auto sign = [p](auto f, auto s[]){ return s[(p&f)!=fs::perms::none]; };
  std::ostringstream os;
  for (auto&& [flag, s] : table) os << sign(flag, s);
  return os.str();
}
```

**Dateieigenschaften** (Tab. 22) können abgefragt werden:

```
auto getfiletime(fs::path path)
{
  auto ftime = fs::last_write_time(path);
#ifdef __MINGW32__ // workaround bis C++20 ?
  auto tp_now  = decltype(ftime)::clock::now();
  auto sys_now = std::chrono::system_clock::now();
  auto systime = sys_now + (ftime - tp_now);
  auto time = std::chrono::system_clock::to_time_t(systime);
#else // Annahme: file_clock = system_clock, nicht für MSVC ...
  auto time = decltype(ftime)::clock::to_time_t(ftime);
#endif
  std::string s(std::asctime(std::localtime(&time)));
  return s.substr(0, s.size()-1); // ohne '\n';
}

void show(fs::path path)
{
  std::cout << (fs::is_directory(path)? "d" : " ")
    << perm_string(status(path).permissions())
    << " " << getfiletime(path)
    << " " << (fs::is_regular_file(path)? fs::file_size(path) : 0)
    << " " << path << '\n';
}
```

[107] Außerdem: `all set_uid set_gid sticky_bit mask unknown` (POSIX-Standard IEEE 1003-2017).

Tabelle 22: Eigenschaften für Dateipfad `p` abfragen (Auswahl).

| | |
|---|---|
| `exists(p)` | Eintrag vorhanden |
| `file_size(p)` | Dateigröße |
| `is_directory(p)` | Verzeichnis? |
| `is_empty(p)` | leer? |
| `is_regular_file(p)` | reguläre Datei? |
| `last_write_time(p)` | wann zuletzt geschrieben? |
| `permissions(p)` | Dateiberechtigungen |
| `space(p)` | freier Speicherplatz im Dateisystem |

**Iteratoren** durchlaufen ein Verzeichnis (mit Unterverzeichnissen):

```
void ls(fs::path p)
{
  for (auto&& entry : fs::directory_iterator(p)) show(entry);
}             // oder fs::recursive_directory_iterator
```

### C.20.2 Verzeichnisse und Dateien anlegen, kopieren, umbenennen, löschen

```
void create()
{
  fs::create_directories("sandbox/a/b");
  fs::create_directories("sandbox/c");
  std::ofstream("sandbox/file1.txt") << "Hello";
  std::ofstream("sandbox/a/file2.txt") << "World";
}

void copy()
{
  fs::rename("sandbox/a/file2.txt", "sandbox/a/file3.txt");
  fs::copy("sandbox/a/file3.txt", "sandbox/c/file3.txt");
  fs::copy("sandbox/a", "sandbox/copy",
    fs::copy_options::recursive |
    fs::copy_options::directories_only);
}

void destroy()
{
  fs::remove("sandbox/file1.txt");
  auto n = fs::remove_all("sandbox");
  std::cout << n << "files/directories deleted\n";
}
```

## C.21 Parallelverarbeitung

### C.21.1 Leichtgewichtige Prozesse

**Threads starten** parallel laufende Arbeitsstränge durch Übernahme einer Funktion:

```
#include <thread>

void task()
{
  std::cout << std::this_thread::get_id() << '\n';
}

void nebenher()
{
  std::thread t{task};
  // anderes zu tun ...
```

Die nebenläufig abzuarbeitende Funktion kann auch ein Funktor oder Lambda-Ausdruck sein. Wertparameter[108] können mitgegeben werden:

```
  std::thread t1{ausgabe, "Gleichzeitig?\n"};
  std::thread t2{ausgabe, "Kann ich nicht.\n"};
  ausgabe("Foyer des Arts\n");
```

**Threads beenden** sich selbst, sobald die übernommene Funktion endet. Der aufrufende Thread (Besitzer) kann warten, bis der von ihm gestartete parallele Ablauffaden abgearbeitet ist und ihn dann zusammenführen oder ihn vorher abkoppeln:[109]

```
  t1.join();    // warten
  t2.detach();  // abkoppeln
  std::this_thread::sleep_for(std::chrono::seconds{42});
}
```

Threads besitzen Verschiebesemantik, sie lassen sich nicht kopieren. Mit der Methode `joinable()` ist erfragbar, ob der Thread noch zusammengeführt werden kann.

Mitunter ist es sinnvoll, die Anzahl der Prozessoren, Kerne bzw. Hyperthreads des Systems mit `std::thread::hardware_concurrency()` zu ermitteln. Wenn dies scheitert, liefert die Funktion 0.

`std::this_thread` erlaubt, den aktiven Thread mit `sleep_for(duration)` für eine vorgegebene Zeitspanne oder mit `sleep_until(time_point)` bis zu einem bestimmten Zeitpunkt schlafen zu legen. Mit `yield()` kann er auf den Rest seiner Zeitscheibe verzichten, damit andere Thread aktiv werden können.

[108] Referenzen auf gemeinsam genutzte Ressourcen werden mit `std::ref(`⟨*Variable*⟩`)` verpackt.
[109] Der Destruktor eines angekoppelten, noch laufenden Threads ruft `std::terminate()` auf.

Tabelle 23: Konstruktorparameter für die Sperren `std::lock_guard<Mutex>` (oberhalb Linie) und `std::unique_lock<Mutex>` (alle).

| | |
|---|---|
| `(mutex)` | übernehmen und sperren |
| `(mutex, std::adopt_lock)` | schon gesperrtes Mutex übernehmen |
| `(mutex, std::defer_lock)` | noch nicht sperren |
| `(mutex, std::try_to_lock)` | sperren, wenn Mutex frei ist |
| `(timed_mutex, timepoint)` | maximal bis zu diesem Zeitpunkt warten |
| `(timed_mutex, duration)` | maximal vorgegebene Dauer warten |
| `(unique_lock)` | Rechtswert-Referenz (Verschiebesemantik) |

### C.21.2 Gemeinsam genutzte Ressourcen

**Wettrennen** (*race conditions*) beim Zugriff auf gemeinsam genutzte Ressourcen führen zu undefiniertem Programmverhalten. Kritische Bereiche werden durch gegenseitigen Ausschluss (*mutual exclusion*) mit `std::mutex`-Variablen vor gleichzeitigem Zugriff durch parallellaufende Threads gesichert. Am Ende des kritischen Bereichs wird die Mutex-Variable wieder freigegeben. Nun kann der nächste Thread diesen Block betreten.

Der Mutex-Typ `std::recursive_mutex` wurde für rekursive Aufrufe entworfen, die Typen `std::timed_mutex` und `std::recursive_timed_mutex` verfügen über Methoden `try_lock_for(duration)` und `try_lock_until(time_point)`, die wie auch `try_lock()` einen Wahrheitswert liefern, ob der Mutex gesperrt werden konnte. Mutex-Methoden kann man selbst aufrufen — es ist aber einfacher, Mutexe über Sperren zu steuern.

**Sperren** `std::lock_guard<Mutex>` bedienen die Methoden `lock()` und `unlock()` der Mutex-Variable in ihrem Konstruktor bzw. Destruktor[110]:

```
#include <mutex>

void ausgabe(std::string s)
{
  static std::mutex mutex;
  std::lock_guard<std::mutex> myLock{mutex};
  std::cout << s << std::endl;
}
```

Die Sperre `std::unique_lock<Mutex>` ist flexibler. Sie bietet größere Freiheiten beim Einrichten (Tab. 23). Die Methoden `lock()`, `unlock()`, `try_lock()` sind zugänglich. Bei eingekapselten `timed_mutex`-Variablen sind die Methoden `try_lock_for(duration)` und `try_lock_until(time_point)` nutzbar. Der Sperrenzustand lässt sich mit `if (myLock)` und `owns_lock()` erfragen. Der Besitz an einem Mutex wird durch `release()` aufgegeben und liefert einen Zeiger auf den Mutex.

[110] RAII-Prinzip, so wird die Sperre auch beim Auftreten von Ausnahmen sicher freigegeben.

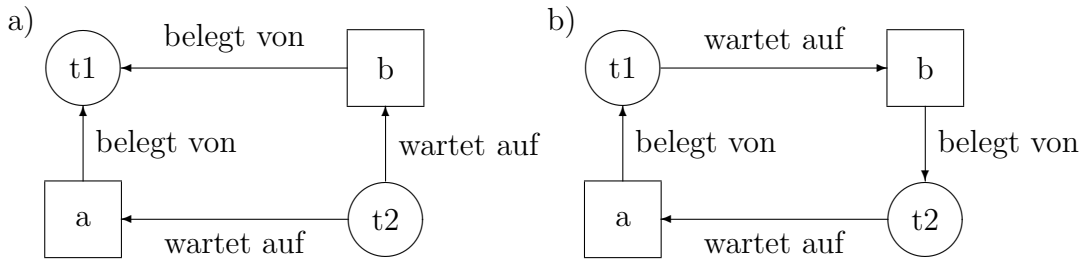



Abbildung 12: Erfolgreiche Ressourcenzuteilung (a) und Verklemmung (b).

**Bedingungsvariablen** `std::condition_variable`[111] können belegte Sperren zeitweilig wieder freigeben, damit andere Threads weiterarbeiten können. Ist die zur Fortführung des Threads notwendige Bedingung erfüllt, kann dieser von einem anderen durch die Methoden `notify_one()` bzw. `notify_all()` wieder erweckt werden, die Sperre wieder erlangen und seine Arbeit fortsetzen:

```
#include <condition_variable>

std::mutex mutex;
std::condition_variable sufficientFunds; // gemeinsam genutzt

void transfer(int from, int to, double amount)
{
  std::unique_lock<std::mutex> myLock{mutex};
  sufficientFunds.wait(myLock, [=]{ return amount <= accounts[from]; });
  accounts[from] -= amount;
  accounts[to]   += amount;
  sufficientFunds.notify_all();
}
```

Dabei ist ⟨*Variable*⟩`.wait(myLock,` ⟨*pred*⟩`);` mit einem Prädikat gleichwertig zu

```
  while (!pred()) variable.wait(myLock);
```

Für `timed_mutex`-Bedingungen sind auch `wait_for(lock, duration,` ⟨*pred*⟩$_{opt}$`)` und `wait_until(lock, time_point,` ⟨*pred*⟩$_{opt}$`)` zulässig.

**Verklemmungen** (*deadlock*) entstehen, wenn mehrere Sperren benötigt werden, die jedoch verschiedenen Threads zugeteilt wurden und durch die zyklische Abhängigkeit keiner der beteiligten Threads fortfahren kann (Abb. 12):

```
  Account a, b;
  std::thread t1{give, std::ref(a), std::ref(b), 5}; // sperrt a, dann b
  std::thread t2{give, std::ref(b), std::ref(a), 5}; // sperrt b, dann a
```

[111] Die allgemeinere `std::condition_variable_any` arbeitet mit jeder Art von Lock oder Mutex.

Die Funktionen `std::lock(`⟨*Sperrenliste*⟩`)` und `std::try_lock(`⟨*Sperrenliste*⟩`)`[112] erlangen alle Sperren der Liste unabhängig von der Reihenfolge ohne Verklemmungsrisiko. Mit der Klasse `std::scoped_lock<Mutexes...>` gelingt das noch einfacher:[113]

```
void give(Account& from, Account& to, double money)
{
  // std::unique_lock<std::mutex> fromLock{from.mutex, std::defer_lock};
  // std::unique_lock<std::mutex> toLock  {to.mutex,   std::defer_lock};
  // std::lock(fromLock, toLock);
  std::scoped_lock lock{from.mutex, to.mutex};  // ab C++17
  from.take(money);
  to.add(money);
}
```

**Threadsichere Initialisierung** globaler Daten erfolgt mit `const_expr`-Konstruktor. Lokale `static`-Variable werden ebenfalls beim ersten Funktionsaufruf threadsicher initialisiert. Betritt ein zweiter Thread die Funktion vor Abschluss der Initialisierung, wird er bis zu deren Ende blockiert. Als `thread_local` gekennzeichnete lokale Variablen existieren unabhängig voneinander in jedem Thread einmal, auch sie werden beim ersten Aufruf im Thread initialisiert. Mit `std::call_once(`⟨*once_flag*⟩`,` ⟨*Funktor*⟩`,` ⟨*Parameter*⟩$_{opt}$`)` können Anfangswertbelegungen verzögert werden:

```
Account a, b;
std::once_flag geschenkt;

void erstausstattung(double amount)
{
  a.add(amount);
  b.add(amount);
}

void work()
{
  std::call_once(geschenkt, erstausstattung, 100.0);
  // transfer random amounts of money
  ...
}
```

Die optionalen Funktionsargumente werden wie bei `std::thread` als Werte übergeben, Referenzen sind durch `std::ref(`⟨*variable*⟩`)` einzukapseln.

[112] Der Rückgabewert ist -1, falls alle Sperren gesetzt werden konnten, sonst die nullbasierte Nummer der fehlgeschlagenen Sperre.

[113] Damit sind nicht alle Möglichkeiten von Verklemmungen beseitigt. Sie können ebenso eintreten, wenn eine erwartete Bedingung nie eintritt oder der wartende Thread nicht benachrichtigt wird. Eine „Liquiditätsklemme“ im obigen Beispiel ist nur durch Bedingungen an das Gesamtsystem vermeidbar.

### C.21.3 Verzögerte Auswertung

**Verzögerte Resultate** `std::future<`⟨*Ergebnistyp*⟩`>` vereinfachen das Anweisen aufwendiger Berechnungen in einem nebenläufigen Prozess:

```
#include <future>

int  frage()  { return 42; }
void ansage() { std::cout << "Die Antwort ist ... "; }

void liefern_auf_bestellung()
{
  std::future<int> antwort = std::async(frage);
  ansage();
  std::cout << antwort.get() << '\n';
}
```

Der Aufruf `std::async(`⟨*Funktor*⟩`,`⟨*Parameter*⟩$_{opt}$`)`[114] startet einen Ablauffaden für die Berechnung eines Wertes. Beim Abruf des Ergebnisses blockiert der Aufrufer, falls die Berechnung noch andauert.

Die Methode `valid()` erfragt, ob das Ergebnis schon vorliegt. Der Wert kann einmalig gelesen werden[115], danach liefert `valid()` wieder `false`, `wait()` blockiert bis zur Fertigstellung der Berechnung, `wait_for(duration)` und `wait_until(time_point)` liefern als Ergebnis einen der Werte `ready`, `deferred` oder `timeout` aus `std::future_status`.

**Aufträge** `std::packaged_task<`⟨*Funktortyp*⟩`>` verpacken eine Funktion, die durch den Klammeroperator mit geeigneten Parametern gestartet wird und liefern das Ergebnis oder die geworfene Ausnahme über ein `std::future<`⟨*Ergebnistyp*⟩`>`. Ein Auftrag kann einem Thread übertragen werden:[116]

```
int frage(int n) { return 6*n; }

void schick_mir_eine_antwort()
{
  std::packaged_task<int(int)> auftrag{frage};
  std::future<int> antwort = auftrag.get_future();
  std::thread t{std::move(auftrag), 7}; // startet: auftrag(7)
  t.detach();
  std::cout << antwort.get() << '\n';
}
```

[114] Bei einer Variante dieses Aufrufs kann als Startverhalten vor dem Funktor `std::launch::async` oder `std::launch::deferred` festgelegt werden. Ohne diese Angabe ist die Entscheidung über das Startverhalten der Implementierung überlassen.

[115] `std::shared_future<T>` erlaubt das mehrmaligen `get()`-Zugriff. Die Methode `share()` erzeugt ein ein solches Objekt und transferiert das Ergebnis zu diesem.

[116] Bei g++ 4.6 kann dies durch einen Implementierungsfehler der Bibliothek nicht übersetzt werden.

**Versprechen** `std::promise<`⟨*Ergebnistyp*⟩`>` dienen mitunter verdeckt als Ablagen für noch zu berechnende Ergebnisse. Im Verbund mit `std::future<`⟨*Ergebnistyp*⟩`>` bilden sie einen Kommunikationskanal zwischen nebenläufigen Prozessen. Der Zugriff auf das künftigen Resultat blockiert, solange kein Wert oder eine Ausnahme hinterlegt wurde:

```
void ansage(std::promise<int>& p)
{
  try
  {
    std::cout << "Die Antwort ist ... ";
    p.set_value(42);
  } catch(...) { p.set_exception(std::current_exception()); }
}

void zur_ablage()
{
  std::promise<int> briefkasten;
  std::future<int> antwort = briefkasten.get_future();
  std::thread t(&ansage, std::ref(briefkasten));
  std::cout << antwort.get() << '\n';
  t.join();
}
```

### C.21.4 Sperrenfreie Kommunikation

`std::atomic<T>` erlauben sicheren sperrenfreien Datenaustausch zwischen Threads:

```
#include <atomic>

int countdown(std::atomic<int>& jobs)
{
  int done = 0, nr;
  while ((nr = int(--jobs)) >= 0) ++done;
  return done;
}

void gemeinsam_gehts_besser()
{
  std::atomic<int> jobs{1000000};
  auto mode = std::launch::async;
  auto cnt1 = std::async(mode, countdown, std::ref(jobs));
  auto cnt2 = std::async(mode, countdown, std::ref(jobs));
  int a = cnt1.get(), b = cnt2.get();
  std::cout << a << " + " << b << " = " << a+b << '\n';
}
```

# D Differenzen

*Have you heard about the new Cray? It's so fast, it executes an infinite loop in 6 seconds.*
– WWW: Computer Jokes

## D.1 Sprachbarrieren

### D.1.1 Plattformabhängigkeiten

Portabilität ist die Eigenschaft, das Programm (als Quelltext) auf eine andere Maschine oder ein anderes Betriebssystem übertragen, übersetzen und ausführen zu können. Portable Programme dürfen keine plattformspezifischen Eigenschaften nutzen. Vereinigungen und Bitfelder vertrauen auf Byte-Reihenfolge (*little/big endian*) und das Ausrichten an Adressen (*byte/word alignment*). Maschinenbefehle und Assembler-Anweisungen (`asm`) sind prozessorabhängig (Registernamen, Befehlssatz). Die Auswertereihenfolge von Ausdrücken ist außer bei `?: && || ,` nicht festgelegt. Compiler können solche Ausdrücke unterschiedlich umsetzen:

```
int i = f() + g(); // erst f(), dann g()?
a[i] = i++;
```

Haben die Funktionen `f()` und `g()` Seiteneffekte, sind Überraschungen beim Plattformwechsel vorprogrammiert. Ausdrücke mit undefiniertem Verhalten, die ja auch für den menschlichen Leser zweideutig sind, sind durch unzweideutige zu ersetzen:

```
int i= f(); i += g(); // Reihenfolge erzwingen!
a[i] = i; i++;
```

Nicht-Standard-Bibliotheken sind nur für bestimmte Plattformen verfügbar. Einige Compilerhersteller packen zusätzliche Funktionen oder Implementierungsdetails in Standardheader, die dort nicht hingehören. Funktionen können Maschineneigenschaften ausnutzen, ohne darauf hinzuweisen. Das kann Portierungen ernsthaft behindern.

### D.1.2 C++ und C

C und C++ sind quellcodeportabel, aber nicht als Bytecode (Objektdatei). Übersetzte Module (Objektdateien) können miteinander verbunden werden. Dies funktioniert jedoch nicht bei verschiedenen Compiler-Herstellern (sofern diese nicht das gleiche Objektformat vereinbart haben), nicht einmal bei verschiedenen Versionen desselben Compilers (Watcom C 9.0/11.0), und natürlich nicht über Plattformgrenzen hinweg (Windows/Linux), selbst wenn vom selben Hersteller.

C-Programme sind mit C++-Compilern übersetzbar, sofern sie keine Eigenschaften von C++ wie Schlüsselwörter nutzen. Strukturnamen sind keine Typen in C — es muss stets `struct name` geschrieben werden. Insofern ist C++ eine Obermenge von C.

Die Bibliotheken von C sind auch mit C++-Compilern nutzbar. Es gibt jedoch C-Quellen, die nicht in C++ übersetzen. Typumwandlungen zwischen Ganzzahlen und Zeigern erfordern in C++ die explizite Typumwandlung (typecast). Ausnahme bilden Zuweisung und Test auf 0 (`NULL`).

Tabelle 24: Compilergenerierte spezielle Methoden ab C++11.

| Nutzer definiert → / Compiler erzeugt | Konstr. | Destr. | Kopie | Zuw. | Verschiebe-konstruktor | Verschiebe-zuweisung |
|---|---|---|---|---|---|---|
| Standardkonstruktor | – | ✓ | – | ✓ | – | ✓ |
| Destruktor | ✓ | □ | ✓! | ✓! | ✓ | ✓ |
| Kopierkonstruktor | ✓ | ✓! | □ | ✓! | × | × |
| Zuweisungsoperator | ✓ | ✓! | ✓! | □ | × | × |
| Verschiebekonstruktor | ✓ | – | – | – | □ | – |
| Verschiebezuweisung | ✓ | – | – | – | – | □ |

Legende: ✓ `= default`, × `= delete`, – = nicht deklariert, ! = Rule of Three beachten.

### D.1.3 Einbinden von Modulen aus C und anderen Sprachen

In C++ aufzurufende C-Funktionen oder C++-Funktionen, die in C-Modulen benutzt werden, sollten mit der Bindung

```
extern "C" int f();
```

deklariert werden. Überladene Funktionen können die `"C"`-Bindung nicht erhalten, über die Modulgrenze hinweg können keine Ausnahmen geworfen werden. Ähnliches gilt für übersetzte Programmteile in Assembler, Fortran und anderen Sprachen:

```
extern "FORTRAN" f();  // IBM C++
```

Die Details sind der Begleitdokumentation des Compilers bzw. Linkers zu entnehmen. Bei Funktionen mit `"C++"`-Bindung werden compilerabhängig Parametertyp-Informationen an den Namen angehangen (*name mangling*), die das Linken mit C-Modulen erschweren.

### D.1.4 C++ von 1998 nach 2011 und später

Compilern kann beim Übersetzen eine bestimmte Sprachversion vorgeschrieben werden (z.B. Option `-std=c++17` oder `-std=c++2a`). Der Implementierungsstand aktueller Compiler ist auf http://en.cppreference.com/w/cpp/compiler_support einzusehen.

**Verschiebesemantik** erlaubt ab C++11 die Unterscheidung zwischen Referenzen auf länger lebende (*lvalue* `X&`) und demnächst abzubauende Objekte (*rvalue* `X&&`). Verschiebekonstruktor und -zuweisung können aus *rvalue-Referenzen* Ressourcen ohne Kopieraufwand übernehmen, z.B. durch die Austausch von Zeigern. Die Regeln für die speziellen Methoden einer Klasse `X` werden damit etwas komplexer (Tab. 24).

## D.2 Geschmacksfragen

*De gustibus non disputandum.*
– Römisches Sprichwort

### D.2.1 Stil

Es gibt eine Menge wohlbegründeter Regeln, wie ein Quelltext aussehen sollte.[117] Oberstes Kriterium sind Lesbarkeit und Verständlichkeit; was „lesbar“ ist, kann sich von Programmierer zu Programmierer unterscheiden.

Der Einsatz von Sprachmitteln liegt ebenso wie die Formatierung im Entscheidungsbereich des Programmierers. Eine Jahrzehnte dauernde Kontroverse über die Programmiersprachen hinweg betraf die Verwendung von `goto`. Seit der Möglichkeit der Ausnahmebehandlung gibt es noch ein Argument mehr dagegen, dennoch wurde `goto` nicht endgültig verbannt. Ein eigener Stil entwickelt sich durch Aneignung, durch Lesen guter Quellen und durch Schreiben.[118]

### D.2.2 Empfehlungen

**Namen** so kurz wie in `x = f(n);` sind kein Zeichen von Faulheit, solange ihr Einsatz örtlich begrenzt erfolgt. Beim Programmieren wie bei anderen schöpferischen Schreibarbeiten tritt ein Informationsstau ein, der durch Kurzschrift bekämpft wird. Längere Namen haben eine größeren Gültigkeitsbereich.

*Beschreibende Namen* `int* reference_counter` sind besser verständlich. Die Zugehörigkeit zu einer Bibliothek wird in C oft durch Präfixe wie in `wxDraw(...)` angezeigt; modernes C++ sollte hierfür Namensbereiche nutzen. *Typcodierende Präfixe* `int* ipReferenceCounter;` verschlüsseln den Typ nochmals im Namensanfang.[119]

Attributnamen in Klassen können besonderen Konventionen folgen: Unterstriche unterstreichen die Rolle als Implementierungsdetails, Präfixe die Zugehörigkeit zur Klasse:

```
struct Vec { double _x, _y, _z; };
struct Person { string its_name; };
```

Lange Namen können mit Unterstrich `a_very_long_name` durchgekoppelt (C-Stil) oder durch Großschreibung `aVeryLongName` unterteilt (Smalltalk-Konvention, *camel casing*) werden.[120] Funktions- und Variablennamen werden traditionell klein geschrieben, Makrokonstanten groß. Eigene Typen können einen großen Anfangsbuchstaben haben.

```
int maximum = RAND_MAX;
struct Vec { int x, y, z; };
```

[117] Allerdings enden solche Empfehlungen erfahrungsgemäß in Kriegen wie bei Gullivers Reise zu den Liliputanern: ob ein Ei am spitzen oder stumpfen Ende aufzuschlagen sei. Der Kompromiss, das Ei in der Mitte aufschlagen, ist unannehmbar: Das Dotter läuft heraus.

[118] Programmierer in Teamarbeit unterliegen ohnehin einem Gruppenzwang.

[119] Die sogenannte *ungarische Notation* wurde vom Microsoft-Programmierer Charles Simonyi für Assembler popularisiert. In objektorientierten Sprachen hat sie nichts verloren.

[120] Die Schreibung in jeweils aktuellen Programmiersprachen beeinflusst die Schrift-Mode: In den 80er Jahren MicroSoft, heute eher intelligent.men@hard.work in Anlehnung an OOP und Webdomänen.

**Initialisierungen** können als Zuweisungen, wie Funktionsaufrufe (Konstruktoren) oder wie Verbunde über Initialisiererlisten formuliert werden. *POD-Typen* (plain old data) werden bei *default initialization* mit Nullen gefüllt, sonst ist ihr Inhalt nicht definiert:

```
int i = 9;  // C                int *p = new int;
int j(9);   // Simula           int *q = new int(); // default init
int k{9};   // C++11            Vec u = { 0 };      // default init
                                Vec v { 1, 2, 3 };
```

**Kommentare** sollten erläutern, was im Code selbst nicht ausdrückbar ist, niemals wiederholen, was schon im Quellcode steht:

```
// BCPL-Stil: Stimmen Quelle und Kommentar nicht ueberein, ...
/* C-Stil:    ... sind vermutlich beide falsch. */
```

Gut zum Deaktivieren geeignet und schachtelbar ist die Präprozessoranweisung:

```
#if 0
  This code won't work ...
#endif
```

**Formatierung** unterstreicht die syntaktische Struktur:

```
class X {     // K&R-Stil       class X      // Allman-Stil
  void f() {                    {
      // ... platzsparend          void f()
  }                                 {
};                                      // ... Schachtelebene
                                    }
                                };
```

Leerzeichen nach Kommas, vor und nach dem Zuweisungsoperator oder auch in mehrstufig geklammerten Ausdrücken können die zu lesenden Blöcke strukturieren:

```
i = j*3 + k/4;
z += y+3;
if (EOF == (c=getchar())) return 0;
```

Der Zeilenumbruch erfolgt in der Regel nach dem Ende einer Anweisung. Er kann in langen Anweisungen die innere Struktur (z. B. Rangfolge von Operationen) betonen:

```
int i =   eine_komplizierte_funktion()
        + eine_variable_mit_sehr_langem_Namen * 25
        - 3;

int mat[3][3] = { 1, 2, 3,
                  4, 5, 6,
                  7, 8, 9 };
```

# Index


`[]`, *siehe* Feld

Ableitung, *siehe* Vererbung
abstrakte Basis, *siehe* Klasse
abstrakte Methode, *siehe* Klasse
Adresse, *siehe* Zeiger
`<algorithm>`, *siehe* Algorithmen
Algorithmen, 92–102
    modifizierend, 96
    mutierend, 98
    nichtmodifizierend, 94
    numerisch, 102
    STL, 92
Aliasname, *siehe* Variable
`and`, *siehe* Operatoren
`and_eq`, *siehe* Operatoren
ANSI C, 1
`<any>`, *siehe* Datenkapsel
`any`, *siehe* Datenkapsel
`argc`, *siehe* Hauptprogramm
`argv`, *siehe* Hauptprogramm
Aufzählungskonstante, 13
Ausdruck, 8
Ausnahme, 56
    erklären, 56
    fangen, 56
    werfen, 56
Auswertereihenfolge, *siehe* Operatoren
`auto`, *siehe* Variable

Basisklasse, *siehe* Vererbung
Bibliothek, *siehe* Modularisierung
Bildschirm, *siehe* Ein-/Ausgabe
`bitand`, *siehe* Operatoren
Bitfeld, *siehe* Struktur, `bitset`
Bitmengen, 114
`bitor`, *siehe* Operatoren
Bitschieben, *siehe* Operatoren
`<bitset>`, *siehe* Bitmengen
Block, 8
`bool`, *siehe* Typ
`break`, *siehe* Schleife, Verzweigung

`case`, *siehe* Verzweigung
`catch`, *siehe* Ausnahme
`cerr`, *siehe* Ein-/Ausgabe
`<cfloat>`, *siehe* Wertebereich
`char`, *siehe* Typ
`<chrono>`, *siehe* Zeit
`cin`, *siehe* Ein-/Ausgabe
`class`, *siehe* Klasse
`<climits>`, *siehe* Wertebereich
`clog`, *siehe* Ein-/Ausgabe
`<cmath>`, *siehe* Mathematik
Compiler, *siehe* Übersetzung
`compl`, *siehe* Operatoren
`<complex>`, *siehe* Mathematik
`<condition_variable>`, *siehe* Thread
`const`, *siehe* Variable, Methode
`const`-Methode, *siehe* Klasse
`const_cast<>`, *siehe* Operatoren
`constexpr`, *siehe* Variable, Methode
Container, 58, 82–87
    -Adapter, 89
    Eigenschaften, 86
    Hash-, 84
    Iterator
        Ein-/Ausgabe-, 91
        zum Einfügen, 91
    Iteratoren, 90–91
    Operationen, 86
        assoziativ, 87
        Hash, 87
        Kosten, 88
        listentypisch, 87
        stapelartig, 87
    sequentiell, 84
    sortiert, 84
        Kriterium, 84
        Schlüssel, 84
    `unordered_...`, 84
`continue`, *siehe* Schleife
copy&swap, *siehe* Operatoren überladen
`cout`, *siehe* Ein-/Ausgabe
`<cstdio>`, *siehe* Ein-/Ausgabe

`<cstring>`, *siehe* Zeichenketten

dangling pointer, *siehe* Zeiger
Dateiarbeit, *siehe* Ein-/Ausgabe
Dateisystem, 121–123
    Berechtigungen, 122
    Iterator, 123
    Pfad, 121
Dateiumlenkung, *siehe* Ein-/Ausgabe
Datenkapseln, 118–120
    any, 118
    optional, 118
    variant, 119
`default`, *siehe* Klasse, Verzweigung
`#define`, *siehe* Vorverarbeitung
Deklarationsdatei, 33
Dekrement, *siehe* Operatoren
`delete`, *siehe* Freispeicher, Klasse
`<deque>`, *siehe* Container
`deque<>`, *siehe* Container
Destruktor, *siehe* Klasse
`directory_iterator`, *siehe* Dateisystem
`do`, *siehe* Schleife
`double`, *siehe* Typ
`duration`, *siehe* Zeitspanne
`dynamic_cast<>`, *siehe* Operatoren

Ein-/Ausgabe, 74–81
    -Umlenkung, 74
    Bildschirm, 74
    C-Funktionen, 81
        Formatanweisung, 81
    Fehlerprotokoll, 74
    Formatierung, 78
        Ausgabebreite, 79
        Manipulatoren, 80
        Nachkommastellen, 79
        Vorzeichen, 79
        Zahlenbasis, 78
    Ströme, 74–77
        Ausgabe, 74
        Dateien, 77
        Eingabe, 75
        mit wahlfreiem Zugriff, 77
        Zeichenketten, 76
        Zustand, 76
    Tastatur, 74
`#elif`, *siehe* Vorverarbeitung
`else`, *siehe* Verzweigung
Endlosschleife, *siehe* Schleife, endlos
Entscheidung, *siehe* Verzweigung
`enum`, *siehe* Aufzählung
Erbe, *siehe* Vererbung
Ergebnistyp, *siehe* Funktion
`#error`, *siehe* Vorverarbeitung
Escape-Sequenz, *siehe* Sonderzeichen
exception, *siehe* Ausnahme
explicit, 49
`explicit`, *siehe* Konstruktor
`extern`, *siehe* Variable, Funktion

`false`, *siehe* Typ
Feld, 20
    Index, 20
    mehrdimensional, 20
    Zeichenkette, 7, 20, 62
`<filesystem>`, *siehe* Dateisystem
`final`, *siehe* Klasse
`float`, *siehe* Typ
`for()`, *siehe* Schleife
Formatierung, *siehe* Ein-/Ausgabe, Stil
`free()`, *siehe* Freispeicher
Freispeicher, 58
    anfordern, 60
    Fehler, 61
    Felder, 60
    freigeben, 60
    Knappheit, 60
    Platzierungssyntax, 61
    smarte Zeiger, 58
Freunde, *siehe* Klasse
`friend`, *siehe* Klasse
`<fstream>`, *siehe* Ein-/Ausgabe
`fstream`, *siehe* Ein-/Ausgabe
`<functional>`, *siehe* Funktoren
Funktion, 30–33
    anonyme, *siehe* Lambda-Ausdruck
    Definition, 30
    Deklaration, 30
    `inline`, 31

Parameter, 11, 32
Feld als, 33
Referenz-, 32
Standard-, 32
Verschiebe-, 32
Wert-, 32
Parameterliste, 30
variabel, 30
Prototyp, 30
Rückkehranweisung, 31
Rumpf, 30
Schablone, 31
überladen, 31
Funktoren, 48, 104–107
Binder, 107
Lambda-Ausdruck, 105
Negierer, 107
Operator-, 106
`<future>`, *siehe* Thread
`future<>`, *siehe* Thread

Ganzzahl, *siehe* Zahl,Typ
Gleitkommazahl, *siehe* Zahl, Typ
`goto`, *siehe* Sprung
`gslice`, *siehe* Wertfelder
Gültigkeitsbereich, *siehe* Variable

Hauptprogramm, 3, 33
Parameter, 33
Rückgabewert, 3
Header, *siehe* Deklarationsdatei
`high_resolution_clock`, *siehe* Uhr

`#if`, *siehe* Vorverarbeitung
`if()`, *siehe* Verzweigung
`#ifdef`, *siehe* Vorverarbeitung
`#ifndef`, *siehe* Vorverarbeitung
`ifstream`, *siehe* Ein-/Ausgabe
`#inlude`, *siehe* Vorverarbeitung
Index, *siehe* Feld
Initialisierung, *siehe* Variable, Klasse
Inkrement, *siehe* Operatoren
`inline`, *siehe* Funktion
Instanz, *siehe* Klasse
`int`, *siehe* Typ
`<iomanip>`, *siehe* Ein-/Ausgabe
`iostream`, *siehe* Ein-/Ausgabe
`<iostream>`, *siehe* Ein-/Ausgabe
`istream`, *siehe* Ein-/Ausgabe
`<iterator>`, *siehe* Iteratoren
Iteratoren, *siehe* Container, Algorithmen

K&R C, 1
Klasse, 42–55
abgeleitete, 51
Basis-, 50
abstrakte, 53
virtuelle, 55
Destruktor, 43
Freunde, 44
Initialisiererliste, 43
Instanz, 42
Komponente, 42
`static`, 44
Konstruktor, 43
-delegation, 43
`default`, 43
`delete`, 43
Initialisierer, 43
Kopier-, 43
Verschiebe-, 43
Methode, 42
abstrakte, 53
`const`, 42
`final`, 53
`inline`, 42
`override`, 53
virtuell, 53
mit Zeiger, 47
Objekt, 42
Schablone, 45
Spezialisierung, 45
Schnittstelle, 42
Vererbung, 50–55
Mehrfach-, 54
nicht-öffentlich, 52
öffentlich, 51
Zugriffsrechte, 44
bei Vererbung, 52
`private`, 44
`protected`, 44

`public`, 44
Koenig lookup, *siehe* Namensraum
Kommentar, 8
Kompatibilität
von C und C++, 130
Komponente, *siehe* Struktur
Konstante, *siehe* Variable, Methode
Konstruktor, *siehe* Klasse
Kopierkonstruktor, *siehe* Klasse
Kurzschrift, *siehe* Zuweisung

Lambda-Ausdruck, *siehe* Funktoren
Lebensdauer, *siehe* Variable
Lesemethode, *siehe* Klasse
`<limits>`, *siehe* Wertebereich
`#line`, *siehe* Vorverarbeitung
Linker, *siehe* Übersetzung
`<list>`, *siehe* Container
`list<>`, *siehe* Container
Literalsuffix, 49
`locale`, 70
logische Verknüpfung, *siehe* Operatoren
`long`, *siehe* Typ

`main()`, *siehe* Hauptprogramm
Makro, *siehe* Vorverarbeitung
`malloc()`, *siehe* Freispeicher
Manipulatoren, *siehe* Ein-/Ausgabe
`<map>`, *siehe* Container
`map<>`, *siehe* Container
`<regex>`, *siehe* regulärer Ausdruck
Mathematik, 108–115
Funktionen, 108
komplexe Zahlen, 108
Tupel, 110
Mehrfachverzweigung, *siehe* Verzweigung
`<memory>`, *siehe* Freispeicher
Methode, *siehe* Klasse
Modularisierung, 38
`multimap<>`, *siehe* Container
`multiset<>`, *siehe* Container
`mutable`, *siehe* Variable
`<mutex>`, *siehe* Thread

Nachkomme, *siehe* Vererbung
Namensraum, 39
Koenig lookup, 40
`namespace`, *siehe* Namensraum
`<new>`, *siehe* Freispeicher
`new`, *siehe* Freispeicher
`not`, *siehe* Operatoren
`not_eq`, *siehe* Operatoren
`nothrow`, *siehe* Freispeicher
`<numeric>`, *siehe* Algorithmen
`numeric_limits<>`, *siehe* Wertebereich

Objekt, *siehe* Klasse
`ofstream`, *siehe* Ein-/Ausgabe
Operationen, *siehe* Operatoren
`operator`, *siehe* Operatoren überladen
Operatoren, 6, 15–19
arithmetisch, 15
Auflistung, 18
Auswertereihenfolge, 19
bitweise, 17
Dekrement, 17
Entscheidung, 17
Inkrement, 17
logische, 17
Methoden, 46
Postfix-, 17
Präfix-, 17
Rangfolge, 19
Typinformation, 18
`sizeof`, 18
`typeid`, 18
Typumwandlung, 16, 18
`const_cast`, 18
`dynamic_cast`, 18
`reinterpret_cast`, 18
`static_cast`, 18
überladen, 46–49
`()`, 48
`[]`, 48
als Funktion, 46
als Methode, 46
Dekrement, 48
Ein-/Ausgabe, 47
Inkrement, 48
Regeln, 49
Typkonverter, 49

Vergleichs-, 17
Zuweisung, 15, 47
RAII, 47, 57
Rule of the Big Three, 47
Verbund-, 15, 46
Operatorobjekte, *siehe* Funktoren
`<optional>`, *siehe* Datenkapsel
`optional<>`, *siehe* Datenkapsel
`or`, *siehe* Operatoren
`or_eq`, *siehe* Operatoren
`ostream`, *siehe* Ein-/Ausgabe
`override`, *siehe* Klasse

Parameter, *siehe* Funktion
`path`, *siehe* Dateisystem
Plattformabhängigkeit, 130
Platzierung, *siehe* Freispeicher
Postfix, *siehe* Operatoren
Präfix, *siehe* Operatoren
`#pragma`, *siehe* Vorverarbeitung
Präprozessor, *siehe* Vorverarbeitung
Priorität, *siehe* Operatoren
`priority_queue<>`, *siehe* Container
`private`, *siehe* Klasse
`promise<>`, *siehe* Thread
`protected`, *siehe* Klasse
Prototyp, *siehe* Funktion
Prozedur, *siehe* Funktion
`public`, *siehe* Klasse

`<queue>`, *siehe* Container
`queue<>`, *siehe* Container

RAII, *siehe* Operatoren überladen
`<random>`, *siehe* Zufallszahlen
Rangfolge, *siehe* Operatoren
`<ratio>`, *siehe* Verhältnisse
`realloc()`, *siehe* Freispeicher
Referenzparameter, *siehe* Funktion
`<regex>`, *siehe* regulärer Ausdruck
`register`, *siehe* Variable
regulärer Ausdruck, 71–73
`reinterpret_cast<>`, *siehe* Operatoren
`return`, *siehe* Funktion
Rückgabetyp, *siehe* Funktion
Rückgabewert, *siehe* Funktion
Rückkehranweisung, *siehe* Funktion
Rule of the Big Five, *siehe* Rule of the Big Three
Rule of the Big Three, *siehe* Operatoren überladen

Schablone, *siehe* Funktion, Klasse
Schleife, 26–27
Abbruch, 27
endlos, *siehe* Endlosschleife
fußgesteuert, 27
kopfgesteuert, 26
range-based for, 105
Testbedingung, 26
Zähl-, 26
Schlüsselwörter, 1
Schlüsselwörter, 6
scope, *siehe* Gültigkeitsbereich
Seiteneffekte, *siehe* Vorverarbeitung
`<set>`, *siehe* Container
`set<>`, *siehe* Container
`short`, *siehe* Typ
`signed`, *siehe* Typ
`sizeof`, *siehe* Operatoren
`slice`, *siehe* Wertfelder
smart pointer, *siehe* Freispeicher
Sonderzeichen, 6, 8
white spaces, 6
Speicherbedarf, *siehe* Typ
Speicherklasse, *siehe* Variable
Sprung, 29
`<stack>`, *siehe* Container
stack unwinding, *siehe* Ausnahme
`stack<>`, *siehe* Container
Standard Template Library, *siehe* Container, Algorithmen
Standardbibliothek
Header, 40
Standardparameter, *siehe* Funktion
`static`, *siehe* Variable, Funktion, Klasse
`static_assert`, *siehe* Variable, Methode
`static_cast<>`, *siehe* Operatoren
`std`, *siehe* Namensraum
Stil, 132
Formatierung, 133

Kommentar, 133
STL, *siehe* Container, Algorithmen
`<string>`, *siehe* `string`-Klasse
`string`-Klasse, 62–69
Ströme, *siehe* Ein-/Ausgabe
`struct`, *siehe* Struktur, Klasse
Struktur, 21
Bitfeld, 22
Komponente, 21
`switch()`, *siehe* Verzweigung
`system_clock`, *siehe* Uhr

Tastatur, *siehe* Ein-/Ausgabe
`template`, *siehe* Funktion, Klasse
Textersatz, *siehe* Vorverarbeitung
`this`, 46
Thread, 124–129
Atomics, 129
Bedingungsvariable, 126
deadlock, 126
Future, 128
Mutex, 125
Promise, 129
race condition, 125
Task, 128
`<thread>`, *siehe* Thread
`throw`, *siehe* Ausnahme
`time_point`, *siehe* Zeitpunkt
`true`, *siehe* Typ
`try`, *siehe* Ausnahme
Typ, 9–10
-erweiterung, 21
Modifikation, 9
Speicherbedarf, 9
`typedef`, 21
Wertebereich, 9
zusammengesetzter, *siehe* Feld, Struktur, Klasse
`typedef`, 45, *siehe* Typ
`typeid`, *siehe* Operatoren
Typparameter, *siehe* Schablone
Typumwandlung, *siehe* Operatoren

überladen, *siehe* Operatoren
Übersetzung, 38

`#undef`, *siehe* Vorverarbeitung
`union`, *siehe* Vereinigung
`<unordered_map>`, *siehe* Container
`unordered_map<>`, *siehe* Container
`unordered_multimap<>`, *siehe* Container
`unordered_multiset<>`, *siehe* Container
`<unordered_set>`, *siehe* Container
`unordered_set<>`, *siehe* Container
`unsigned`, *siehe* Typ
Unterprogramm, *siehe* Funktion
`using`, *siehe* Namensraum

`<valarray>`, *siehe* Wertfelder
Variable, 11–13
Definition, 11
Deklaration, 11
flüchtig, 13
Gültigkeitsbereich, 11
global, 11
lokal, 11
Konstante, 13
Lebensdauer, 12
automatisch, 12
statisch, 12
Referenz, 12
Speicherklasse, 12
`extern`, 12
`static`, 12
statisch lokal, 12
Typ, 11
veränderbar, 13
`<variant>`, *siehe* Datenkapsel
`variant<>`, *siehe* Datenkapsel
`<vector>`, *siehe* Container
`vector<>`, *siehe* Container
Verbundzuweisung, *siehe* Zuweisung
Vereinigung, 22
Vererbung, *siehe* Klasse
Vergleich, *siehe* Operatoren
Verhältnisse, 110
Verschiebekonstruktor, *siehe* Klasse
Verzweigung, 28–29
Entscheidung, 28
Mehrfach-, 28
`virtual`, *siehe* Klasse

virtuell, *siehe* Klasse
virtuelle Basis, *siehe* Klasse
virtuelle Methode, *siehe* Klasse
`void`, *siehe* Typ
`volatile`, *siehe* Variable
Vorfahr, *siehe* Vererbung
Vorverarbeitung, 8, 34–37
  bedingte Übersetzung, 37
  Einbinden von Dateien, 37
  Makro
    -Funktion, 34
    -Konstante, 34
    Umwandlung in Zeichenkette, 36
    vordefiniert, 36

Wertebereich, *siehe* Typ
Wertfelder, 114
Wertparameter, *siehe* Funktion
`while()`, *siehe* Schleife
white spaces, *siehe* Sonderzeichen
Wiederholung, *siehe* Schleife
wild pointer, *siehe* Zeiger

`xor`, *siehe* Operatoren
`xor_eq`, *siehe* Operatoren

Zahlkonstanten, 7
  ganz, 7
  Gleitkomma-, 7
Zählschleife, *siehe* Schleife
Zeichen, 7
Zeichenkette, *siehe* Feld, `string`-Klasse
Zeiger, 23–25
  -arithmetik, 24
  Adresse, 23
  auf Funktion, 25
  auf Struktur, 25
  Differenz, 24
  Feldnamen, 24
  Komponentenzugriff, 25
  Vergleich, 24
  wilde, 23
Zeit, 116–117
  Uhr, 116
  Zeitpunkt, 117
  Zeitspanne, 116
Zufallszahlen, 111–112
Zugriffsrecht, *siehe* Klasse
Zugriffsschutz, *siehe* Klasse
zusammengesetzte Typen, *siehe* Typ