

C++ programmieren

C++ lernen – professionell anwenden –
Lösungen nutzen

» Hier geht's
direkt
zum Buch

DIE LESEPROBE

■ 1.4 Das erste Programm

Das klassische erste Programm ist ein Mini-Programm, das einfach nur »Hello World!« ausgibt. Das Listing 1.1 zeigt den Programmcode.

Listing 1.1: Hello World-Programm (*cppbuch/k1/hello.cpp*)

```
#include <iostream>

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

Die Entwicklung eines einfachen Programms lernen Sie hier an einer ebenfalls einfachen Aufgabe kennen: Es sollen zwei Zahlen addiert werden. Dabei wird Ihnen zunächst das Programm vorgestellt und gleich danach erfahren Sie, wie Sie es eingeben und zum Laufen bringen können. Der erste Schritt besteht in der Formulierung der Aufgabe. Sie lautet: »Lies zwei Zahlen a und b von der Tastatur ein. Berechne die Summe beider Zahlen und zeige das Ergebnis auf dem Bildschirm an.« Die Aufgabe ist so einfach, wie sie sich anhört! Im zweiten Schritt wird die Aufgabe in die Teilaufgaben »Eingabe«, »Berechnung« und »Ausgabe« zerlegt:

Listing 1.2: Programmentwurf

```
int main()                // Noch tut dieses Programm nichts!
{
    // Lies zwei Zahlen ein
    /* Berechne die Summe beider
       Zahlen */
    // Zeige das Ergebnis auf dem Bildschirm an
}
```

Sie sehen einen einfachen Entwurf, der gleichzeitig ein C++-Programm ist. Es tut allerdings noch nichts. Es bedeuten:

```
int           ganze Zahl zur Rückgabe
main          Name der Funktion, mit der jedes Programm beginnt
( )          Innerhalb dieser Klammern können der Funktion Informationen
            mitgegeben werden.
{ }          Block
/* ... */    Kommentar, der über mehrere Zeilen gehen kann
// ...      Kommentar bis Zeilenende
```

Ein durch { und } begrenzter *Block* enthält die Anweisungen an den Rechner. Der *Compiler* übersetzt den Programmtext in eine rechnerverständliche Form. Im Block sind lediglich *Kommentare* enthalten und noch keine Anweisungen an den Computer, sodass unser Programm nichts tut.

Kommentare werden einschließlich der Kennungen vom Compiler vollständig ignoriert. Ein Kommentar, der mit /* beginnt, wird mit der ersten */-Zeichenkombination beendet,

auch wenn er sich über mehrere Zeilen erstreckt. Ein mit // beginnender Kommentar endet am Ende der Zeile. Auch wenn Kommentare ignoriert werden, sind sie doch sinnvoll für alle, die ein Programm lesen. Die Anweisungen zu erläutern hilft denjenigen, die Ihre Nachfolge antreten, weil Sie befördert worden sind oder die Firma verlassen haben. Kommentare sind auch wichtig für den Autor eines Programms, der ohne sie nach einem halben Jahr nicht mehr weiß, warum er gerade diese oder jene komplizierte Anweisung geschrieben hat. Sie sehen:

Ein Programm ist ein Text!

- Der Text hat eine Struktur entsprechend den C++-Sprachregeln: Es gibt Wörter wie hier das Schlüsselwort `main`. In C++ werden alle Schlüsselwörter kleingeschrieben. Es gibt weiterhin Zeilen, Satzzeichen und Kommentare.
- Die Bedeutung des Textes wird durch die Zeilenstruktur nicht beeinflusst. Mit \ und folgendem **ENTER** ist eine Worttrennung am Zeilenende möglich. Das Zeichen \ wird »Backslash« genannt. Mit dem Symbol **ENTER** ist hier und im Folgenden die Betätigung der großen Taste **↵** rechts auf der Tastatur gemeint.
- Groß- und Kleinschreibung werden unterschieden! `main()` ist nicht dasselbe wie `Main()`.

Weil die Zeilenstruktur für den Rechner keine Rolle spielt, kann der Programmtext nach Gesichtspunkten der Lesbarkeit gestaltet werden. Im dritten Schritt müssen nur noch die Inhalte der Kommentare als C++-Anweisungen formuliert werden. Dabei bleiben die Kommentare zur Dokumentation stehen, wie im Beispielprogramm unten zu sehen ist.



Hinweis

Alle Programmbeispiele sind von der Internet-Seite <http://cppbuch.de/> herunterladbar. In den Listings finden Sie den zugehörigen Dateinamen in der Überschrift oder in der ersten Zeile des Listings.

Listing 1.3: Summe zweier Zahlen berechnen (*cppbuch/k1/summe.cpp*)

```
#include <iostream>
using namespace std;

int main()
{
    int summand1 {0};
    int summand2 {0};
    // Lies zwei Zahlen ein
    cout << "_Zwei_ganze_Zahlen_eingeben:";
    cin >> summand1 >> summand2;
    /* Berechne die Summe beider Zahlen
    */
    int summe = summand1 + summand2;
    // Zeige das Ergebnis auf dem Bildschirm an
    cout << "Summe=" << summe << '\n';
    return 0;
}
```

Es sind einige neue Worte dazugekommen, die hier kurz erklärt werden. Machen Sie sich keine Sorgen, wenn Sie nicht alles auf Anhieb verstehen! Alles wird im Verlauf des Buchs wieder aufgegriffen und vertieft. Wie das Programm zum Laufen gebracht wird, werden Sie bald erfahren.

<code>#include<iostream></code>	Einbindung der Ein-/Ausgabefunktionen. Diese Zeile muss in jedem Programm stehen, das Eingaben von der Tastatur erwartet oder Ausgaben auf den Bildschirm bringt. Sie können sich vorstellen, dass der Compiler beim Übersetzen des Programms an dieser Stelle erst alle zur Ein- und Ausgabe notwendigen Informationen liest. Details folgen in Abschnitt 2.3.
<code>using namespace std;</code>	Der Namensraum (englisch <i>namespace</i>) <code>std</code> wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Erklärungen folgen auf den Seiten 63 und 151.
<code>int main()</code>	<code>main()</code> ist die Funktion, mit der jedes Programm beginnt (es gibt auch andere Funktionen). Der zu <code>main()</code> gehörende Programmcode wird durch die geschweiften Klammern <code>{</code> und <code>}</code> eingeschlossen. Ein mit <code>{</code> und <code>}</code> begrenzter Bereich heißt <i>Block</i> . Mit <code>int</code> ist gemeint, dass die <code>main()</code> -Funktion nach Beendigung eine Zahl vom Typ <code>int</code> (= ganze Zahl) an das Betriebssystem zurückgibt. Dazu dient die unten beschriebene <code>return</code> -Anweisung. Normalerweise – das heißt bei ordnungsgemäßem Programmablauf – wird die Zahl 0 zurückgegeben. Andere Zahlen können über das Betriebssystem einen Fehler signalisieren.
<code>int summand1 {0};</code> <code>int summand2 {0};</code> <code>int summe = ...</code>	<i>Deklaration</i> (Bekanntmachung) von Objekten: Mitteilung an den Compiler, der ab jetzt die Namen <code>summand1</code> , <code>summand2</code> und <code>summe</code> innerhalb des Blocks <code>{ }</code> kennt. Hier wird gleichzeitig Speicherplatz bereitgestellt. Es gibt verschiedene Zahlentypen in C++. Mit <code>int</code> sind ganze Zahlen gemeint: <code>summe</code> , <code>summand1</code> , <code>summand2</code> sind ganze Zahlen. Oft ist es sinnvoll, einen Anfangswert festzulegen, etwa 0, wie hier bei <code>summand1</code> und <code>summand2</code> .
<code>;</code>	Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe weiter unten).
<code>cin</code>	Eingabe: Der Doppelpfeil zeigt hier in Richtung des Objekts, das ja von der Tastatur einen neuen Wert aufnehmen soll. Die Information fließt von der Eingabe <code>cin</code> zum Objekt <code>summand1</code> beziehungsweise zum Objekt <code>summand2</code> .
<code>cout</code>	Ausgabe: <code>cout</code> (Abkürzung für <i>character out</i> oder <i>console out</i>) ist die Standardausgabe. Der Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe <code>cout</code> gesendet wird, zum Beispiel <code>cout << summand1;</code> . Wenn mehrere Dinge ausgegeben werden sollen, sind sie durch <code><<</code> zu trennen.

"Text"	beliebige Zeichenkette, die die Anführungszeichen selbst nicht enthalten darf, weil sie als Anfangs- beziehungsweise Endmarkierung einer Zeichenfolge dienen. Wenn die Zeichenfolge die Anführungszeichen enthalten soll, sind diese als \" zu schreiben: cout << \"C++\" ist der Nachfolger von \"C\"!"; erzeugt die Bildschirmausgabe "C++" ist der Nachfolger von "C"!
'\n'	Die Ausgabe des Zeichens \n bewirkt eine neue Zeile.
return 0;	Unser Programm läuft einwandfrei, es gibt daher 0 an das Betriebssystem zurück. Diese Anweisung darf in der main()-Funktion fehlen, dann wird automatisch 0 zurückgegeben.

<iostream> ist ein Header. Dieser aus dem Englischen stammende Begriff (head = dt. Kopf) drückt aus, dass Zeilen dieser Art am Anfang eines Programmtextes stehen. Der Begriff wird im Folgenden verwendet, weil es keine gängige deutsche Entsprechung gibt. Einen Header mit einem Dateinamen gleichzusetzen, ist meistens richtig, nach dem C++-Standard aber nicht zwingend. *Hinweis:* Tatsächlich wird ein Programm zunächst von einem vorgeschalteten *Präprozessor* bearbeitet, der das Ergebnis der Bearbeitung an den eigentlichen Compiler weiterleitet. Wenn im Folgenden also von »Compiler« die Rede ist, ist meistens auch der Präprozessor gemeint.

summand1, summand2 und summe sind veränderliche Daten und heißen Variablen. Sie sind Objekte eines vordefinierten Grunddatentyps für ganze Zahlen (int), mit denen die üblichen Ganzzahloperationen wie + und - durchgeführt werden können. Der Begriff »Variable«³ wird für ein veränderliches Objekt gebraucht. Für Variablen gilt:

- Sie müssen deklariert werden. int summe; ist eine Deklaration, wobei int der *Datentyp* des Objekts summe ist, der die Eigenschaften beschreibt. Entsprechendes gilt für summand1 und summand2. Die Objektnamen sind frei wählbar im Rahmen der unten angegebenen Grammatikregeln. Unter *Deklaration* wird verstanden, dass der Name dem Compiler bekannt gemacht wird. Wenn dieser Name danach im Programm versehentlich falsch geschrieben wird, kennt der Compiler den falschen Namen nicht und gibt eine Fehlermeldung aus. Somit dienen Deklarationen der Programmsicherheit.
- Objektnamen bezeichnen Bereiche im Speicher des Computers, deren Inhalte verändert werden können. Die Namen sind symbolische Adressen, unter denen der Wert gefunden wird. Über den Namen kann dann auf den aktuellen Wert zugegriffen werden (siehe Abbildung 1.1).

Der Speicherplatz wird vom Compiler reserviert. Man spricht dann von der *Definition* der Objekte. Definition und Deklaration werden unterschieden, weil es auch Deklarationen ohne gleichzeitige Definition gibt, doch davon später mehr. Zunächst sind die Deklarationen zugleich Definitionen. Abbildung 1.2 zeigt den Ablauf der Erzeugung eines lauffähigen Programms. Ein Programm ist ein Text, von Menschenhand geschrieben (über Programmgeneratoren soll hier nicht gesprochen werden) und dem Rechner unverständlich. Um dieses Programm auszuführen, muss es erst vom Compiler in eine für den Computer verständliche Form übersetzt werden.

³ Anmerkung für Menschen mit Vorkenntnissen: Aus reiner C++-Sicht ist eine Variable eine Deklaration eines Objekts (oder einer Referenz) und sagt nichts darüber aus, ob es konstant oder veränderlich ist. Die Eigenschaft »konstant« wird durch das Schlüsselwort const bewirkt.

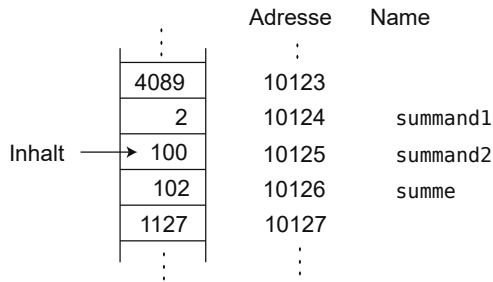


Abbildung 1.1: Speicherbereiche mit Adressen

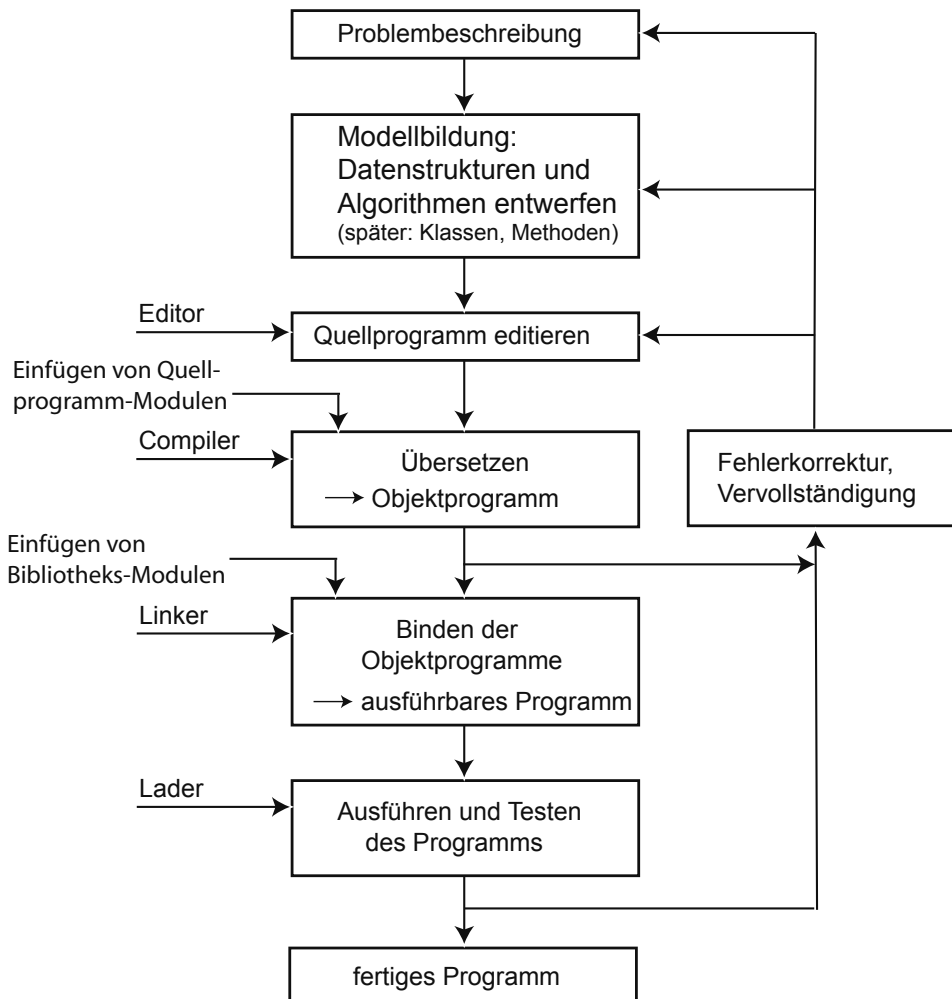


Abbildung 1.2: Erzeugung eines lauffähigen Programms

Der Compiler ist selbst ein Programm, das bereits in maschinenverständlicher Form vorliegt und speziell für diese Übersetzung zuständig ist. Nach Eingabe des Programmtextes mit dem Editor können Sie den Compiler starten.

Ein Programmtext wird auch »Quelltext« oder »Quellcode« (englisch *source code*) genannt. Der Compiler erzeugt aus dem Quellcode den Objektcode, der noch nicht ausführbar ist. Hinter den einfachen Anweisungen `cin >> ...` und `cout << ...` verbergen sich eine Reihe von Aktivitäten wie die Abfrage der Tastatur und die Ansteuerung des Bildschirms, die nicht speziell programmiert werden müssen, weil sie schon in vorübersetzter Form in Bibliotheksdateien vorliegen. Die Aufrufe dieser Aktivitäten im Programm müssen mit den dafür vorgesehenen Algorithmen in den Bibliotheksdateien zusammengebunden werden, eine Aufgabe, die der *Linker* übernimmt, auch *Binder* genannt. Der Linker bindet Ihren Objektcode mit dem Objektcode der Bibliotheksdateien zusammen und erzeugt daraus ein ausführbares Programm, das nun gestartet werden kann. Der Aufruf des Programms bewirkt, dass der *Lader*, eine Funktion des Betriebssystems, das Programm in den Rechner Speicher lädt und startet. Diese Schritte werden stets ausgeführt, auch wenn sie in den Programmentwicklungsumgebungen verborgen ablaufen. Bibliotheksmodule können auch während der Programmausführung geladen werden (nicht im Bild dargestellt).

Wie bekomme ich ein Programm zum Laufen?

Nachdem Sie den Programmtext mit einem Editor geschrieben haben, speichern Sie ihn als Datei *summe.cpp* ab. Öffnen Sie nun unter Windows eine Konsole (cmd-Eingabeaufforderung oder PowerShell) bzw. ein Terminal unter Linux oder macOS, und wechseln mit `cd` in das Verzeichnis, wo Sie *summe.cpp* abgespeichert haben. Die Übersetzung, auch *Compilation* genannt, wird mit

```
g++ -o summe.exe summe.cpp
```

gestartet. Das Programm wird durch Eintippen von *summe.exe* (oder *./summe.exe*, wenn das aktuelle Verzeichnis nicht im Pfad ist) gestartet. Eigentlich verbergen sich hinter dem Aufruf des Compilers zwei Schritte:

```
g++ -c summe.cpp           compilieren (summe.o wird erzeugt)
g++ -o summe.exe summe.o   linken
g++ -o summe.exe summe.cpp beide Schritte zusammengefasst
```

Die Objektdateien können je nach System die Endung *.o* oder *.obj* tragen. Wenn `g++ -std=c++23 -o summe.exe summe.cpp` geschrieben wird, soll der Compiler den C++-Standard 2023 verwenden. Das ist bei diesem einfachen Programm nicht notwendig, weil es keine der neueren Eigenschaften nutzt.

Eine integrierte Entwicklungsumgebung lässt die genannten Schritte per Tastendruck ablaufen, wie Sie gleich sehen.

switch/case-lokale Variablen

Wenn in der case-Anweisung eine lokale Variable angelegt wird, muss sie in einem eigenen Gültigkeitsbereich (scope) mit geschweiften Klammern {} gekapselt werden:

```
case 'D':
    zahl = 500;
    {
        auto x{zahl*10};           // Beginn des lokalen Gültigkeitsbereichs.
        cout << x << '\n';
    }                             // Die Gültigkeit von x endet hier.
    break;
```

Das Weglassen der geschweiften Klammern führt zu einer Fehlermeldung des Compilers. Wie bei der if-Anweisung ist es möglich, eine lokale Variable für die switch()-Anweisung anzulegen. Die Variable wird in den runden Klammern initialisiert, also etwa `switch(int x=1; auswahl)`.

1.8.5 Wiederholungen

Häufig muss die gleiche Teilaufgabe oft wiederholt werden. Denken Sie nur an die Summation von Tabellenspalten in der Buchführung oder an das Suchen einer bestimmten Textstelle in einem Buch. In C++ gibt es zur Wiederholung von Anweisungen drei verschiedene Arten von Schleifen. In einer Schleife wird nach Abarbeitung einer Teilaufgabe (zum Beispiel Addition einer Zahl) wieder an den Anfang zurückgekehrt, um die gleiche Aufgabe noch einmal auszuführen (Addition der nächsten Zahl). Durch bestimmte Bedingungen gesteuert, zum Beispiel Ende der Tabelle, bricht irgendwann die Schleife ab.

Schleifen mit while

Abbildung 1.8 zeigt die Syntax von while-Schleifen.

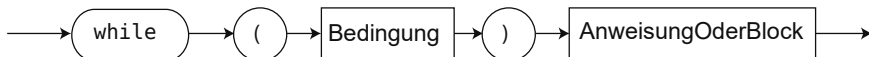
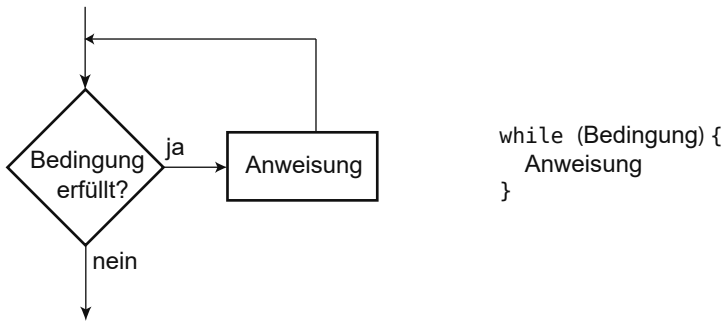


Abbildung 1.8: Syntaxdiagramm einer while-Schleife

AnweisungOderBlock ist wie auf Seite 66 definiert. Die Bedeutung einer while-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis true oder ungleich 0 liefert, wird die Anweisung bzw. der Block ausgeführt. Die Bedingung wird auf jeden Fall zuerst geprüft. Wenn die Bedingung von vornherein unwahr ist, wird die Anweisung gar nicht erst ausgeführt (siehe Abbildung 1.9).

Die Anweisung oder der Block innerhalb der Schleife heißt *Schleifenkörper*. Schleifen können wie if-Anweisungen beliebig geschachtelt werden.

```
while (Bedingung1)           // geschachtelte Schleifen, ohne und mit geschweiften Klammern
    while (Bedingung2) {
        .....
        while (Bedingung3) {
            .....
        }
    }
}
```

```

while (Bedingung) {
  Anweisung
}
  
```

Abbildung 1.9: Flussdiagramm für eine `while`-Anweisung

Beispiele

■ Unendliche Schleife:

```

while (true)
  Anweisung
  
```

■ Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```

while (false)
  Anweisung
  
```

■ Summation der Zahlen 1 bis 99:

```

int sum {0};
int n {1};
constexpr int grenze {99};
while (n <= grenze) {
  sum += n++;
}
  
```

■ Berechnung des größten gemeinsamen Teilers $\text{ggT}(x, y)$ für zwei natürliche Zahlen x und y nach Euklid. Es gilt:

- $\text{ggT}(x, x)$, also $x == y$: Das Resultat ist x .
- $\text{ggT}(x, y)$ bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also $\text{ggT}(x, y) == \text{ggT}(x, y-x)$, falls $x < y$.

Das Ersetzen der Differenz geschieht im folgenden Beispiel iterativ, also durch eine Schleife.

Listing 1.15: Beispiel für `while`-Schleife (*cppbuch/k1/ggt.cpp*)

```

#include <iostream>
using namespace std;

int main()
{
  int x {0};
  int y {0};
  cout << "2_Zahlen_>_0_eingeben_:";
  cin >> x >> y;
  cout << "Der_ggT_von_" << x << "_und_" << y << "_ist_";
  while (x != y) {
  
```

```

    if (x > y) {
        x -= y;
    }
    else {
        y -= x;
    }
}
cout << x << '\n';
}

```

Innerhalb einer Schleife muss es eine Veränderung derart geben, dass die Bedingung irgendwann einmal unwahr wird, sodass die Schleife abbricht (man sagt auch *terminiert*). Unbeabsichtigte »unendliche« Schleifen sind ein häufiger Programmierfehler. Im ggT-Beispiel ist leicht erkennbar, dass die Schleife irgendwann beendet sein *muss*:

1. Bei jedem Durchlauf wird mindestens eine der beiden Zahlen kleiner.
2. Die Zahl 0 kann nicht erreicht werden, da immer eine kleinere von einer größeren Zahl subtrahiert wird. Die *while*-Bedingung schließt die Subtraktion gleich großer Zahlen aus, und nur die könnte 0 ergeben.

Daraus allein ergibt sich, dass die Schleife beendet wird, und zwar in weniger als x Schritten, wenn x die anfangs größere Zahl war. Im Allgemeinen sind es erheblich weniger, wie eine genauere Analyse ergibt.



Tipp

Die Anweisungen zur Veränderung der Bedingung sollen möglichst an das Ende des Schleifenkörpers gestellt werden, um sie leicht finden zu können.

Schleifen mit *do while*

Abbildung 1.10 zeigt die Syntax einer *do while*-Schleife.

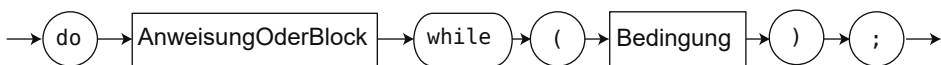


Abbildung 1.10: Syntaxdiagramm einer *do while*-Schleife

AnweisungOderBlock ist wie auf Seite 66 definiert. Die Anweisung oder der Block einer *do while*-Schleife wird ausgeführt, und *erst anschließend* wird die Bedingung geprüft. Ist sie wahr, wird die Anweisung ein weiteres Mal ausgeführt usw. Die Anweisung wird also mindestens einmal ausgeführt. Im Flussdiagramm ist die Anweisung ein Block (siehe rechts in der Abbildung 1.11).

do while-Schleifen eignen sich unter anderem gut zur sicheren Abfrage von Daten, indem die Abfrage so lange wiederholt wird, bis die abgefragten Daten in einem plausiblen Bereich liegen, wie im Primzahlprogramm unten zu sehen ist. Es empfiehlt sich zur besseren Lesbarkeit, *do while*-Schleifen strukturiert zu schreiben. Die schließende geschweifte Klammer soll genau unter dem ersten Zeichen der Zeile stehen, die die öffnende geschweifte Klammer enthält. Dadurch und durch Einrücken des dazwischen stehenden Textes ist sofort der Schleifenkörper erkennbar.

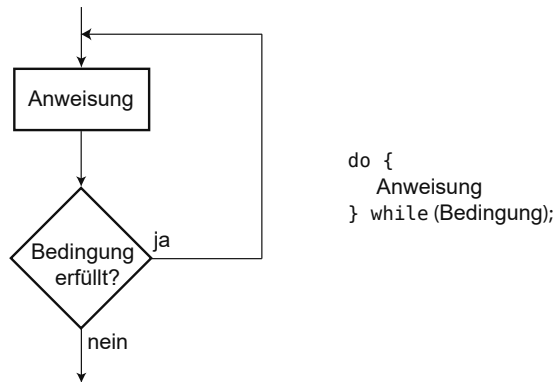


Abbildung 1.11: Flussdiagramm für eine `do while`-Anweisung

```
do {
    Anweisungen
} while (Bedingung);
```

Das *direkt hinter* die abschließende geschweifte Klammer geschriebene `while` macht unmittelbar deutlich, dass dieses `while` zu einem `do` gehört. Das ist besonders wichtig, wenn der Schleifenkörper in einer Programmliste über die Seitengrenze ragt. Eine `do while`-Schleife kann stets in eine `while`-Schleife umgeformt werden (und umgekehrt).

Listing 1.16: Berechnen einer Primzahl mit `do while` (*cppbuch/k1/primzahl.cpp*)

```
#include <cmath>
#include <iostream>
using namespace std;

int main()
{
    cout << "Berechnung_der_ersten_Primzahl,_die_>="
         << "_der_eingegebenen_Zahl_ist\n";
    // Hinweis: Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
    long zahl {0L};
    // do while-Schleife zur Eingabe und Plausibilitätskontrolle
    do {
        // Abfrage, solange zahl ≤ 3 ist
        cout << "Zahl_>_3_eingeben_:";
        cin >> zahl;
    } while (zahl <= 3);

    if (zahl % 2 == 0) { // Falls zahl gerade ist, wird die nächste
                       // ungerade Zahl als Startwert genommen.
        ++zahl;
    }
    bool gefunden {false};
    do {
        // limit = Grenze, bis zu der gerechnet werden muss.
        // sqrt() arbeitet mit double, daher wird der Typ explizit umgewandelt.
        const long limit {1 + static_cast<long>(sqrt(static_cast<double>(zahl)))};
        long rest {0L};
```

```

long teiler {1L};
do { // Kandidat zahl durch alle ungeraden Teiler dividieren
    teiler += 2;
    rest = zahl % teiler;
} while (rest > 0 && teiler < limit);

if (rest > 0 && teiler >= limit) {
    gefunden = true;
}
else { // sonst nächste ungerade Zahl untersuchen
    zahl += 2;
}
} while (!gefunden);
cout << "Die_nächste_Primzahl_ist_" << zahl << '\n';
}

```



Tipp

In do-while-Schleifen wird die Bedingung erst am Ende abgefragt, sie ist beim Lesen also nicht sofort zu finden. Deswegen verwenden Sie do-while nur, wenn Sie erreichen wollen, dass eine Schleife mindestens einmal durchlaufen werden soll.

Schleifen mit for

Die letzte Art von Schleifen ist die for-Schleife. Sie wird häufig eingesetzt, wenn die Anzahl der Wiederholungen vorher feststeht, aber das muss durchaus nicht so sein. Abbildung 1.12 zeigt die Syntax einer for-Schleife.

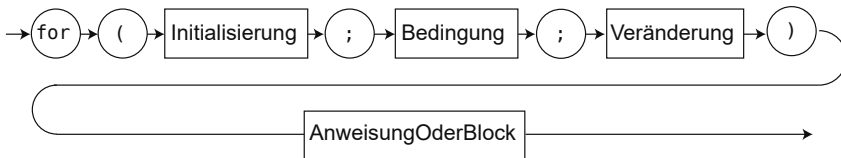


Abbildung 1.12: Syntaxdiagramm einer for-Schleife

Der zu wiederholende Teil (Anweisung oder Block) wird auch Schleifenkörper genannt. Beispiel: ASCII-Tabelle im Bereich 65 ... 69 ausgeben

```

for (int i = 65; i <= 69; ++i) {
    cout << i << "_" << static_cast<char>(i) << '\n';
}

```

Bei der Abarbeitung werden die folgenden Schritte durchlaufen:

1. Durchführung der Initialisierung, zum Beispiel Startwert für eine Laufvariable festlegen. Eine Laufvariable wird wie *i* in der Beispielschleife als Zähler benutzt.
2. Prüfen der Bedingung.
3. Falls die Bedingung wahr ist, zuerst die Anweisung und dann die Veränderung ausführen.

Die Laufvariable `i` kann auch außerhalb der runden Klammern deklariert werden, dies gilt aber als schlechter Stil. Der Unterschied besteht darin, dass außerhalb der Klammern deklarierte Laufvariablen noch über die Schleife hinaus gültig sind.

```
int i; // nicht empfohlen
for (i = 0; i < 100; ++i) {
    // Programmcode, i ist hier bekannt
}
// i ist weiterhin bekannt ...
```

Im Fall der Deklaration innerhalb der runden Klammern bleibt die Gültigkeit auf den Schleifenkörper beschränkt:

```
for (int i = 0; i < 100; ++i) { // empfohlen
    // Programmcode, i ist hier bekannt
}
// i ist hier nicht mehr bekannt
```

Die zweite Art erlaubt es, `for`-Schleifen als selbstständige Programmteile hinzuzufügen oder zu entfernen, ohne Deklarationen in anderen Schleifen ändern zu müssen. Derselbe Mechanismus gilt für Deklarationen in den runden Klammern von `if`- und `switch`-Anweisungen.

Listing 1.17: Beispiel für `for`-Schleife (*cppbuch/k1/fakultaet.cpp*)

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Fakultät_berechnen._Zahl_>=_0?: ";
    int n {0};
    cin >> n;

    long fak {1L};
    for (int i = 2; i <= n; ++i) {
        fak *= i;
    }
    cout << n << "!_==_" << fak << '\n';
}
```

Verändern Sie niemals die Laufvariable innerhalb des Schleifenkörpers! Das Auffinden von Fehlern würde durch die Änderung erschwert.

```
for (int i = 65; i < 70; ++i) {
    // eine Seite Programmcode
    --i; // irgendwo dazwischen erzeugt eine unendliche Schleife
    // noch mehr Programmcode
}
```

Auch wenn der Schleifenkörper nur aus einer Anweisung besteht, wird empfohlen, ihn in geschweiften Klammern `{ }` einzuschließen.

Äquivalenz von for und while

Eine for-Schleife entspricht direkt einer while-Schleife, sie ist im Grunde nur eine Umformulierung, solange nicht continue vorkommt, das im folgenden Abschnitt beschrieben wird:

```
for (Initialisierung; Bedingung; Veraenderung)
    Anweisung
```

ist äquivalent zu:

```
{
    Initialisierung;
    while (Bedingung) {
        Anweisung
        Veraenderung;
    }
}
```

Die äußeren Klammern sorgen dafür, dass in der Initialisierung deklarierte Variablen wie bei der for-Schleife nach dem Ende nicht mehr gültig sind. Anweisung kann wie immer auch eine Verbundanweisung (Block) sein, in der mehrere Anweisungen stehen können, durch geschweifte Klammern begrenzt. Die umformulierte Entsprechung des obigen Beispiels (ASCII-Tabelle von 65 ... 69 ausgeben) lautet:

```
{
    int i {65}; // Initialisierung
    while (i < 70) { // Bedingung
        cout << i << " " << static_cast<char>(i) << '\n'; // Anweisung
        ++i; // Veränderung
    }
}
```

float- oder double-Laufvariablen vermeiden!

Wegen der Rechenungenauigkeit kann es bei nicht-integralen Typen wie double oder float zu nicht vorhersagbarem Verhalten kommen. Das Beispiel:

```
for (double d = 0.4; d <= 1.2; d += 0.4) {
    cout << d << '\n';
}
```

lässt auf den ersten Blick die Ausgabe 0.4, 0.8, 1.2 erwarten, tatsächlich werden auf meinem System nur die zwei Zahlen 0.4 und 0.8 ausgegeben. Wenn ich jedoch

```
for (double d = 0.5; d <= 1.5; d += 0.5) {
    cout << d << '\n';
}
```

ausführe, werden wie erwartet die drei Zahlen 0.5, 1 und 1.5 angezeigt. Der Grund liegt darin, dass 0.5 im Binärsystem exakt darstellbar ist, 0.4 jedoch nicht. Schon ein Unterschied im letzten Bit lässt den Vergleich auf Gleichheit scheitern. Ganz ungünstig kann sich die Prüfung auf Ungleichheit mit != auswirken:

Die Ausgabe des Programms ist:

```
Objekt 0 wird erzeugt.  
main wird begonnen  
Objekt 1 wird erzeugt.  
    neuer Block  
    Objekt 2 wird erzeugt.  
    Block wird verlassen  
    Objekt 2 wird zerstört.  
main wird verlassen  
Objekt 1 wird zerstört.  
Objekt 0 wird zerstört.
```

Der Destruktor von Objekten mit statischer Lebensdauer (`static` oder globale Objekte) wird nicht nur beim Verlassen eines Programms mit `return`, sondern auch beim Verlassen mit `exit()` aufgerufen. Im Gegensatz zum normalen Verlassen eines Blocks wird der Speicherplatz bei `exit()` jedoch nicht freigegeben.

■ 3.7 Wie kommt man zu Klassen und Objekten? Ein Beispiel

Es kann hier keine allgemeine Methode gezeigt werden, wie man von einer Aufgabe zu Klassen und Objekten kommt. Es wird jedoch anhand eines Beispiels ein erster Eindruck vermittelt, wie der Weg von einer Problemstellung zum objektorientierten Programm aussehen kann.

Es geht hier um ein Programm, das zu einer gegebenen Personalnummer den Namen heraus sucht. Ähnlichkeiten mit der Aufgabe 1.25 von Seite 112 sind beabsichtigt. Gegeben sei eine Datei `daten.txt` mit den Namen und den Personalnummern der Mitarbeiter. Dabei folgt auf eine Zeile mit dem Namen eine Zeile mit der Personalnummer. Das `#`-Zeichen ist die Endekennung. Der Inhalt der Datei ist:

```
Hans Nerd  
06325927  
Juliane Hacker  
19236353  
Michael Ueberflieger  
73643563  
#
```

Einige Analyse-Überlegungen

Um die Problemstellung zu verdeutlichen, wird sie aus verschiedenen Blickwinkeln betrachtet. Es handelt sich dabei nur um *Möglichkeiten*, nicht um den einzig wahren Lösungsansatz (den es nicht gibt).

1. In der Analyse geht es zunächst einmal darum, den typischen Anwendungsfall (englisch *use case*) *in der Sprache des (späteren Programm-)Anwenders* zu beschreiben. Ein ganz konkreter Anwendungsfall, Szenario genannt, ist ein weiteres Hilfsmittel zum Verständnis dessen, was das Programm tun soll.
2. Im zweiten Schritt wird versucht, beteiligte Objekte, ihr Verhalten und ihr Zusammenwirken zu identifizieren.



Anwendungsfall (use case)

Das Programm wird gestartet. Alle Namen und Personalnummern werden zur Kontrolle ausgegeben (weil es hier nur wenige sind). Anschließend erfragt das Programm eine Personalnummer und gibt daraufhin den zugehörigen Namen aus oder aber die Meldung, dass der Name nicht gefunden wurde. Die Abfrage soll beliebig oft möglich sein. Wird X oder x eingegeben, beendet sich das Programm.

Für einen konkreten Anwendungsfall (= Szenario) wird die oben dargestellte Datei *daten.txt* verwendet.



Szenario

Das Programm wird gestartet und gibt aus:

```
Hans Nerd 06325927
Juliane Hacker 19236353
Michael Ueberflieger 73643563
```

Anschließend erfragt das Programm eine Personalnummer. Die Person vor dem Bildschirm (User) gibt 19236353 ein. Das Programm gibt Juliane Hacker aus und fragt wieder nach einer Personalnummer. Jetzt wird 99999 eingegeben. Das Programm meldet nicht gefunden! und fragt wieder nach einer Personalnummer. Jetzt wird X eingegeben. Das Programm beendet sich.

Objekte und Operationen identifizieren

Im nächsten Schritt wird versucht, die beteiligten Objekte und damit ihre Klassen zu identifizieren und eine Beschreibung ihres Verhaltens zu finden.

In der nicht-objektorientierten Lösung zur Vorläuferaufgabe 1.25 werden alle Aktivitäten in `main()` abgehandelt. Das ist unvorteilhaft, weil die Funktionalität damit nicht einfach in ein anderes Programm transportiert werden kann. Deswegen bietet es sich an, die Aktivitäten in ein eigens dafür geschaffenes Objekt zu verlegen. Die Klasse dazu sei hier etwas hochtrabend *Personalverwaltung* genannt. Was müsste so ein Objekt tun?

1. Die Datei *daten.txt* lesen und die gelesenen Daten speichern. Der Einfachheit halber wird hier angenommen, dass keine andere Datei zur Auswahl steht.
2. Die Daten auf dem Bildschirm *ausgeben*.
3. Einen *Dialog* mit dem Benutzer *führen*, in dem nach der Personalnummer gefragt wird.

Diese drei Punkte und die Kenntnis der Datei führen zu entsprechenden Schlussfolgerungen. Dabei sind im ersten Schritt die Substantive (Hauptworte) als Kandidaten für Klassen

zu sehen und Verben (Tätigkeitsworte) als Methoden. Passivkonstruktionen sollen dabei vorher stets in Aktivkonstruktionen verwandelt werden, d.h. *ausgeben* ist besser als *die Ausgabe erfolgt*.

1. Eine Auswahl der Datei ist hier nicht vorgesehen. Ein Objekt der Klasse `Personalverwaltung` soll daher schon beim Anlegen die Datei einlesen und die Daten speichern. Das übernimmt am besten der Konstruktor, dem der Dateiname übergeben wird.
 - Die gelesenen Daten gehören zu Personen. Jede *Person* hat einen Namen und eine Personalnummer. Es bietet sich an, Name und Personalnummer in einer Klasse `Person` zu kapseln. Aus Gründen der Einfachheit sollen Vor- und Nachname nicht getrennt gehalten werden; ein Name genügt.
 - Die Personalnummer soll nicht als `int` vorliegen, sondern als `string`, damit nicht führende Nullen (siehe Datei oben) beim Einlesen verschluckt werden oder zu einer Interpretation als Oktalzahl führen. Außerdem könnte es Nummernsysteme mit Buchstaben und Zahlen geben.
 - Die Klasse `Personalverwaltung` soll die Daten speichern. Dafür bietet sich ein `vector<Person>` als Attribut an.
2. Das Tätigkeitswort *ausgeben* legt nahe, eine gleichnamige Methode `ausgeben()` vorzusehen. In der Methode werden Name und Personalnummer einer Person ausgegeben. Es muss also entsprechende Methoden in der Klasse `Person` geben, etwa `getName()` und `getPersonalnummer()`. Diese Methoden würden innerhalb der Funktion `ausgeben()` aufgerufen werden.
3. *Dialog führen* legt nahe, eine Methode `dialogfuehren()` oder kurz `dialog()` vorzusehen.

Weil nur ein erster Eindruck vermittelt werden soll und die Problemstellung einfach ist, wird auf eine vollständige objektorientierte Analyse (OOA) und ein entsprechendes Design (OOD) verzichtet und auf die Literatur verwiesen, die die OOA/D-Thematik behandelt, zum Beispiel [Oe]. In diesem einfachen Fall konzentrieren wir uns gleich auf eine Lösung mit C++. Eine `main()`-Funktion könnte wie folgt aussehen:

Listing 3.36: `main`-Programm zur Personalverwaltung (`cppbuch/k3/personalverwaltung/main.cpp`)

```
#include "Personalverwaltung.h"
#include <iostream>
using namespace std;

int main()
{
    Personalverwaltung personalverwaltung("daten.txt");
    cout << "Gelesene_Namen_und_Personalnummern:\n";
    personalverwaltung.ausgeben();
    personalverwaltung.dialog();
    cout << "Programmende\n";
}
```

Die Klasse `Person` ist einfach zu entwerfen:

Listing 3.37: Klasse `Person` (`cppbuch/k3/personalverwaltung/Person.h`)

```
#ifndef PERSON_H
```

```

#define PERSON_H
#include <string>

class Person {
public:
    Person(const std::string& name_, const std::string& personalnummer_)
        : name{name_}, personalnummer{personalnummer_}
    { }

    [[nodiscard]] auto getName() const { return name; }

    [[nodiscard]] auto getPersonalnummer() const { return personalnummer; }

private:
    std::string name;
    std::string personalnummer;
};
#endif

```

Auch die Klasse Personalverwaltung ist nach den obigen Ausführungen nicht schwierig, wenn man sich zunächst auf die Prototypen der Methoden beschränkt:

Listing 3.38: Klasse Personalverwaltung (*cppbuch/k3/personalverwaltung/Personalverwaltung.h*)

```

#ifndef PERSONALVERWALTUNG_H
#define PERSONALVERWALTUNG_H
#include "Person.h"
#include <vector>

class Personalverwaltung {
public:
    explicit Personalverwaltung(const std::string& dateiname);

    void ausgeben() const;

    void dialog() const;

private:
    std::vector<Person> personal;
};
#endif

```

Für die Implementierung der Methoden der Klasse Personalverwaltung muss man sich mehr Gedanken machen. Das überlasse ich Ihnen (siehe die nächste Aufgabe)! Die Lösung dürfte aber nicht schwer sein, wenn Sie die Aufgabe 1.25 von Seite 112 gelöst oder deren Lösung nachgesehen haben.



Übungen

3.5 Implementieren Sie die oben deklarierten Methoden der Klasse Personalverwaltung in einer Datei *Personalverwaltung.cpp*.

3.6 Wie können Sie mit C++ erreichen, dass ein Attribut *direkt*, also ohne Einsatz einer Methode, zwar gelesen, aber nicht verändert werden kann? Beispiel:

22

Von der UML nach C++

Dieses Kapitel behandelt die folgenden Themen:

- Vererbung
- Interfaces
- Assoziationen
- Multiplizität
- Aggregation
- Komposition

Die Unified Modeling Language (UML) ist eine weit verbreitete grafische Beschreibungssprache für Klassen, Objekte, Zustände, Abläufe und noch mehr. Sie wird vornehmlich in der Phase der Analyse und des Softwareentwurfs eingesetzt. Auf die UML-Grundlagen wird hier nicht eingegangen; dafür gibt es gute Bücher wie [Oe]. Hier geht es darum, die wichtigsten UML-Elemente aus Klassendiagrammen in C++-Konstruktionen, die der Bedeutung des Diagramms möglichst gut entsprechen, umzusetzen. Die vorgestellten C++-Konstruktionen sind Muster, die als Vorlage dienen können. Diese Muster sind nicht einzigartig, sondern nur Empfehlungen, die Umsetzung zu gestalten. Im Einzelfall kann eine Variation sinnvoll sein.

22.1 Vererbung

Über Vererbung als »ist ein«-Beziehung wurde in diesem Buch schon einiges gesagt, was hier nicht wiederholt werden muss. Sie finden alles dazu in Kapitel 6. Die Abbildung 22.1 zeigt das zugehörige UML-Diagramm.

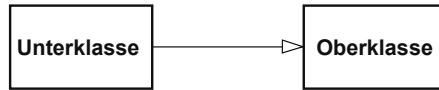


Abbildung 22.1: Vererbung (»ist ein«-Beziehung)

In vielen Darstellungen wird die Oberklasse oberhalb der abgeleiteten Unterklasse dargestellt; in der UML ist aber nur der Pfeil mit dem Dreieck entscheidend, nicht die relative Lage. In C++ wird Vererbung syntaktisch durch »: public« ausgedrückt:

Listing 22.1: Syntaktische Repräsentation der Vererbung

```
class Unterklasse : public Oberklasse {
    // ... Rest weggelassen
};
```

22.2 Interface anbieten und nutzen

Interface anbieten

Abbildung 22.2 zeigt das zugehörige UML-Diagramm. Die Klasse Anbieter implementiert das Interface Schnittstelle-X. Bei der Vererbung stellt die abgeleitete Klasse die Schnittstelle der Oberklasse zur Verfügung. Insofern gibt es eine Ähnlichkeit, auch gekennzeichnet durch die gestrichelte Linie im Vergleich zum vorherigen Diagramm.



Abbildung 22.2: Interface-Anbieter

Die Ähnlichkeit wird in der Umsetzung nach C++ abgebildet: Anbieter wird von dem Interface SchnittstelleX¹ abgeleitet. Um klarzustellen, dass es um ein Interface geht, soll SchnittstelleX abstrakt sein. Das Datenobjekt d wird nicht als const-Referenz übergeben, weil service() damit auch die Ergebnisse an den Aufrufer übermittelt. Ein einfaches Programmbeispiel finden Sie im Verzeichnis *cppbuch/k22/interface*.

¹ Die UML erlaubt Bindestriche in Namen, C++ nicht.

Listing 22.2: Schnittstellenklasse

```

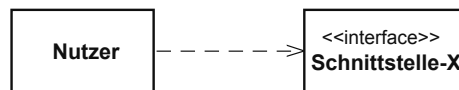
class SchnittstelleX {
public:
    virtual void service(Daten& d) = 0;        // abstrakte Klasse
    virtual ~SchnittstelleX() = default;     // virtueller Destruktor
    SchnittstelleX() = default;
    SchnittstelleX(const SchnittstelleX&) = delete;
    SchnittstelleX& operator=(const SchnittstelleX&) = delete;
};

class Anbieter : public SchnittstelleX {
public:
    void service(Daten& d)
    {
        // ... Implementation der Schnittstelle
    }
};

```

Interface nutzen

Bei der Nutzung des Interfaces bedient sich der Nutzer einer entsprechenden Methode des Anbieters. Die Abbildung 22.3 zeigt das zugehörige UML-Diagramm.

**Abbildung 22.3:** Interface-Nutzer

Ein Nutzer muss ein Anbieter-Objekt kennen, damit der Service genutzt werden kann. Aus diesem Grund wird in der folgenden Klasse bereits dem Konstruktor von Nutzer ein Anbieter-Objekt übergeben, und zwar per Referenz, nicht per Zeiger. Der Grund: Zeiger können nullptr sein, aber undefinierte Referenzen gibt es nicht.

Listing 22.3: Nutzer der Schnittstelle

```

class Nutzer {
public:
    Nutzer(SchnittstelleX& a)
    : anbieter(a)
    {
        daten = ...
    }

    void nutzen()
    {
        anbieter.service(daten);
    }
private:
    Daten daten;
    SchnittstelleX& anbieter;
};

```

Warum wird die Referenz oben nicht als `const` übergeben? Das kann je nach Anwendungsfall sinnvoll sein oder auch nicht. Es hängt davon ab, ob sich der Zustand des Anbieter-Objekts durch den Aufruf der Funktion `service(daten)` ändert. Wenn ja, zum Beispiel durch interne Protokollierung der Aufrufe, entfällt `const`.

22.3 Assoziation

Eine Assoziation sagt zunächst einmal nur aus, dass zwei Klassen in einer Beziehung (mit Ausnahme der Vererbung) stehen. Die Art der Beziehung und zu wie vielen Objekten sie aufgebaut wird, kann variieren. In der Regel gelten Assoziationen während der Lebensdauer der beteiligten Objekte. Nur kurzzeitige Verbindungen werden meistens nicht notiert. Ein Beispiel für eine kurzzeitige Verbindung ist der Aufruf `anbieter.service(daten)`; `anbieter` kennt durch die Parameterübergabe das Objekt `daten`, wird aber vermutlich die Verbindung nach Ablauf der Funktion lösen.

Einfache gerichtete Assoziation

Die Abbildung 22.4 zeigt das UML-Diagramm einer einfachen gerichteten Assoziation.



Abbildung 22.4: Gerichtete Assoziation

Mit »gerichtet« ist gemeint, dass die Umkehrung nicht gilt, wie zum Beispiel die Beziehung »ist Vater von«. Falls zwar Klasse1 die Klasse2 kennt, aber nicht umgekehrt, wird dies durch ein kleines Kreuz bei Klasse1 vermerkt. Es kann natürlich sein, dass eine Beziehung zwischen zwei Objekten *derselben* Klasse besteht. Im UML-Diagramm führt dann der von einer Klasse ausgehende Pfeil auf dieselbe Klasse zurück. In C++ wird eine einfache gerichtete Assoziation durch ein Attribut `zeigerAufKlasse2` realisiert:

Listing 22.4: Gerichtete Assoziation: Klasse1 kennt Klasse2

```

class Klasse1 {
public:
    Klasse1()
    : zeigerAufKlasse2(nullptr)
    { }

    void setKlasse2(Klasse2* ptr2)
    {
        zeigerAufKlasse2 = ptr2;
    }
private:
    Klasse2* zeigerAufKlasse2;
};
  
```

Ein Zeiger ist hier besser als eine Referenz geeignet, weil es sein kann, dass das Kennenlernen erst nach dem Konstruktoraufruf geschieht.

Gerichtete Assoziation mit Multiplizität

Die Multiplizität, auch Kardinalität genannt, gibt an, zu wie vielen Objekten eine Verbindung aufgebaut werden kann. In Abbildung 22.5 bedeutet die 1, dass jedes Objekt der Klasse2 zu genau einem Objekt der Klasse1 gehört. Das Sternchen * bei Klasse2 besagt, dass einem Objekt der Klasse1 beliebig viele Objekte der Klasse2 zugeordnet sind, also möglicherweise auch keins.

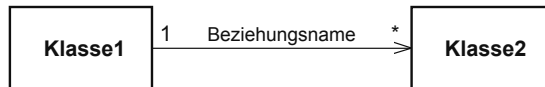


Abbildung 22.5: Gerichtete Assoziation mit Multiplizitäten

Im folgenden C++-Beispiel entspricht Fan der Klasse1 und Popstar der Klasse2. Ein Fan kennt N Popstars. Die Beziehung ist also »kennt«. Der Popstar hingegen kennt seine Fans im Allgemeinen nicht. Um die Multiplizität auszudrücken, bietet sich ein vector an, der Verweise auf Popstar-Objekte speichert. Wenn die Verweise eindeutig sein sollen, ist ein set die bessere Wahl.

Listing 22.5: Gerichtete Assoziation mit Multiplizität: Ein Fan kennt Popstars, aber nicht umgekehrt.

```

class Fan {
public:
    void werdeFanVon(Popstar* star)
    {
        meineStars.insert(star);           // einfügen
    }

    void denKannsteVergessen(Popstar* star)
    {
        meineStars.erase(star);          // entfernen. Rückgabewert ignoriert
    }
    // Rest weggelassen

private:
    std::set<Popstar*> meineStars;
};
  
```

Die Objekte als Kopie abzulegen, also Popstar als Typ für den Set statt Popstar* zu nehmen, hat Nachteile. Erstens ist es wenig sinnvoll, die Kopie zu erzeugen, wenn es doch das Original gibt, und zweitens kostet es Speicherplatz und Laufzeit. Es gibt nur einen Vorteil: Es könnte ja sein, dass es das originale Popstar-Objekt nicht mehr gibt, zum Beispiel durch ein delete irgendwo. Ein noch existierender Zeiger wäre danach auf eine undefinierte Speicherstelle gerichtet. Eine noch existierende Kopie könnte als Wiedergänger auftreten.

Einfache ungerichtete Assoziation

Eine ungerichtete Assoziation wirkt in beiden Richtungen und heißt deswegen auch bidirektionale Assoziation. Die Abbildung 22.6 zeigt das UML-Diagramm.



Abbildung 22.6: Ungerichtete Assoziation

Wenn zwei sich kennenlernen, kann das mit einer ungerichteten Assoziation modelliert werden. Zur Abwechslung sei die Umsetzung in C++ nicht mit zwei, sondern nur mit einer Klasse (namens `Person`) gezeigt. Das heißt, die Klasse hat eine Beziehung zu sich selbst, siehe Abbildung 22.7. Solche Assoziationen werden auch rekursiv genannt und dienen zur Darstellung der Beziehung verschiedener Objekte derselben Klasse.

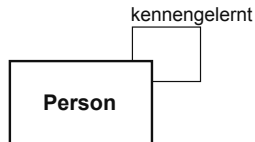


Abbildung 22.7: Rekursive Assoziation

Die Umsetzung in C++ wird am Beispiel von Personen gezeigt, die sich gegenseitig kennenlernen. Ein Aufruf `A.lerntkennen(B)`; impliziert, dass `B` auch `A` kennenernt. Natürlich kann es vorkommen, dass es zwei Personen mit demselben Namen gibt, hier `Frau Holle`.

Listing 22.6: Assoziation: Personen lernen sich kennen (*cppbuch/k22/bidirektAssoziation/main.cpp*)

```

#include "Person.h"

int main()
{
    Person mabuse("Dr._Mabuse");
    Person klicko("Witwe_Klicko");
    Person holle1("Frau_Holle");
    Person holle2("Frau_Holle");           // eine Namensvetterin!
    mabuse.lerntkennen(klicko);
    holle1.lerntkennen(klicko);
    holle1.lerntkennen(holle2);
    mabuse.bekannteZeigen();
    klicko.bekannteZeigen();
    holle1.bekannteZeigen();
}
  
```

Die entscheidende Methode der Klasse `Person` ist `lerntkennen(Person& p)` (siehe unten). Beim Eintrag in die Menge der Bekannten wird festgestellt, ob der Eintrag vorher schon vorhanden war. Wenn nicht, wird er auch auf der Gegenseite vorgenommen. Wenn in der Menge der Bekannten nur die Namen als `String` gespeichert würden, könnten sich zwei Personen mit demselben Namen nicht kennenlernen. Deswegen werden die Adressen der `Person`-Objekte gespeichert (siehe Attribut `set<Person*>` in der Klasse unten).

Listing 22.7: Klasse Person (*cppbuch/k22/bidirektAssoziation/Person.h*)

```

#ifndef PERSON_H
#define PERSON_H
#include <iostream>
#include <set>
#include <string>
#include <utility>

class Person {
public:
    Person(std::string name_)
        : name(std::move(name_))
    {}

    auto getName() const
    {
        return name;
    }

    void lerntkennen(Person& p)
    {
        bool nichtvorhanden = bekannte.insert(&p).second; // siehe Hinweis im Text
        if (nichtvorhanden) { // falls unbekannt, auch bei p eintragen
            p.lerntkennen(*this);
        }
    }

    void bekannteZeigen() const
    {
        std::cout << "Die_Bekanntes_von_" << getName() << "_sind:\n";
        for (const auto& bekannt : bekannte) {
            std::cout << bekannt->getName() << '\n';
        }
    }

private:
    std::string name;
    std::set<Person*> bekannte;
};
#endif

```

**Hinweis**

Die Methode `insert()` eines Sets gibt ein `pair`-Objekt zurück. Das ist eine Struktur mit den Elementen `first` und `second`. `second` ist ein Wahrheitswert, der angibt, ob das Einfügen stattgefunden hat. Wenn nein, war das Element schon vorhanden. `first` ist ein Iterator auf das Element, ob gerade eingefügt oder schon vorhanden gewesen.

■ 22.3.1 Aggregation

Die »Teil-Ganzes«-Beziehung (englisch *part of*) wird auch *Aggregation* genannt. Sie besagt, dass ein Objekt aus mehreren Teilen besteht (die wiederum aus Teilen bestehen können). Die Abbildung 22.8 zeigt das UML-Diagramm. Die Struktur entspricht der gerichteten Assoziation, sodass deren Umsetzung in C++ hier Anwendung finden kann. Ein Teil kann für sich allein bestehen, also auch vom Ganzen gelöst werden. Letzteres geschieht in C++ durch Nullsetzen des entsprechenden Zeigers.

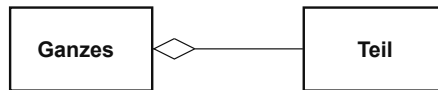


Abbildung 22.8: Aggregation

■ 22.3.2 Komposition

Die Komposition ist eine spezielle Art der Aggregation, bei der die Existenz der Teile vom Ganzen abhängt. Damit ist gemeint, dass die Teile zusammen mit dem Ganzen erzeugt und auch wieder vernichtet werden. Ein Teil ist somit stets genau einem Ganzen zugeordnet; die Multiplizität kann also nur 1 sein. Formal ist auch 0 erlaubt. Für ein isoliertes Objekt ist jedoch der Begriff »Teil« nicht sinnvoll. Die Abbildung 22.9 zeigt das UML-Diagramm.

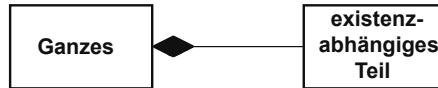


Abbildung 22.9: Komposition

Es empfiehlt sich, bei der Umsetzung in C++ Werte statt Zeiger zu nehmen. Dann ist gewährleistet, dass die Lebensdauer der Teile an das Ganze gebunden ist:

Listing 22.8: Umsetzung der Komposition

```

class Ganzes {
public:
    Ganzes(int datenFuerTeil1, int datenFuerTeil2)
        : ersterTeil(datenFuerTeil1),
          zweiterTeil(datenFuerTeil2)
    {
        // ...
    }
    // ...

private:
    Teil ersterTeil;
    Teil zweiterTeil;
};
  
```

Register

Symbole

* 45, 218, 463
*= 45, 383
+, +=, -, -= 45
++ 45, 385, 459, 463
, 59, 81, 519
-> 221
->* 265
-- 45, 388
.* 265
... *siehe* Ellipse
/, /= 45
/* ... */ 30
// 30
:: 62, 180, 315
::* 265
; 32, 70
<, <=, >, >= 45, 57
<=> 45, 398
<< 45, 107, 258, 371, 427
<<=, >>= 45
=, == 45, 57
== 372
>> 45, 105, 258, 425, 449
?: 70
[] 222, 225, 377
[][]
Matrixklasse 415

Zeigerdarstellung 250

142
%, %= 45
& Adress-Operator 124
&, &= Bit-Operatoren 45
&& logisches UND 57
&& Shell 658
&& R-Wert 480
\ 31, 55, 140
\0 227, 231–233
\", \a, \b 55
\f, \n, \r, \t, \v 55, 105
\x 55
\\ 55
~ 45
^ 45
|= 45
|| 57
! 45, 57
!= 57, 463
\$<, \$^, \$@ 655
@D, @F 665
" 33, 227, 228

A

abgeleitete Klasse 302, 310, 314
und virtueller Destruktor 326
Abhängigkeit (make) 652

- automatische Ermittlung 657
- abort() 955
- abs() 51, 789, 790, 953, 954
- abstrakte Klasse 319, 977
- abstrakter Datentyp 175, 977
- accumulate() 737
- Achterbahnzahlen 98
- acos() 790, 953
- acosh() 790
- Adapter, Iterator- 899
- Additionsoperator 369
- adjacent_difference() 742
- adjacent_find() 772
- adjustfield 435
- Adresse 218
 - symbolische 33
- Adressoperator 219
- advance() 897
- Aggregat 89, 224, 862
- Aggregation 712, 977
- Aktualparameter 118
- <algorithm> 713, 803, 905
- Algorithmus 28, 459
 - accumulate() 737
 - adjacent_difference() 742
 - adjacent_find() 772
 - all_of, any_of 751
 - binary_search() 774
 - clamp() 812
 - copy(), copy_backward() 800
 - copy_if(), copy_n() 802
 - count(), count_if() 750
 - equal() 788
 - equal_range() 776
 - exclusive_scan() 741
 - fill(), fill_n() 735
 - find(), find_if(), find_if_not() 767
 - find_end() 771
 - find_first_of() 768
 - for_each(), for_each_n() 799
 - generate(), _n() 736
 - includes() 778
 - inclusive_scan() 740
 - inner_product() 739
 - inplace_merge() 766
 - iota() 737
 - is_heap(), is_heap_until() 786
 - is_partitioned() 758
 - is_permutation() 754
 - is_sorted(), is_sorted_until() 760
 - iter_swap() 803
 - lexicographical_compare() 755
 - lexicographical_compare_three_way() 756
 - lower_bound() 775
 - make_heap() 784
 - make_pair() 836
 - make_tuple() 837
 - max(...) 810
 - max_element() 743
 - merge() 765
 - mergesort() 766
 - min(...) 810
 - min_element() 743
 - minmax() 811
 - minmax_element() 743
 - mismatch() 786
 - move(), move_backward() 834
 - next_permutation() 753
 - none_of 751
 - nth_element() 764
 - partial_sort(), _copy 762
 - partial_sum() 740
 - partition() 757
 - partition_copy(), partition_point() 758
 - pop_heap() 783
 - prev_permutation() 752
 - push_heap() 784
 - remove(), _if(), _copy(), _copy_if() 807
 - replace(), _if(), _copy(), _copy_if() 806
 - reverse(), reverse_copy() 749
 - rotate(), rotate_copy() 744
 - sample() 749
 - search() 770, 771
 - search_n() 773
 - set_difference() 780
 - set_intersection() 779
 - set_symmetric_difference() 781
 - set_union() 779

- shuffle() 746
- sort() 760
- sort_heap() 785
- split() 714
- stable_partition() 757
- stable_sort() 761
- swap(), swap_ranges() 803
- transform() 804
- transform_exclusive_scan() 741
- transform_inclusive_scan() 741
- unique(), unique_copy() 746
- upper_bound() 775
- Alias 124, 219, 221
 - *this 244
- alignment_of, alignof() 527
- all_of 751
- allgemeiner Konstruktor 184
- allocator 830
- alternative Funktions-Syntax 283
- and_then() 364
- Anführungszeichen 33, 227, 228
- Anker (bei regulärem Ausdruck) 538
- anonymer Namespace 154
- Anweisung 64
- any() (Bitset) 842
- any_of 751
- app 446
- append()
 - Pfad 818
 - String 934
- apply() 839
- Äquivalenz 399, 775
 - klasse 674
- arg() 789, 790
- argc 243
- Argument *siehe* Parameter
- Argument Dependent Lookup (ADL) 978
- argv[] 243
- Arithmetik
 - mit Iteratoren 462
 - mit Zeigern 224
- arithmetische Operatoren 45
- Arität (Template) 512
- Array
 - char 229
 - von C-Strings 229
 - dynamisches 234, 374
 - Freigabe 238
 - als Funktionsparameter 240, 248
 - vs. Zeiger 223
 - zweidimensionales
 - Matrixklasse 415, 416
- <array> 852, 862
- array 95, 862
- Array2d 251
- ASCII 915
 - Dateien 258
 - Tabelle 55, 959
- asctime() 958
- asin() 790, 953
- asinh() 790
- assert() 143
- assign() 858, 935
- assignable_from 534
- Assoziation (UML) 708
- Assoziativität von Operatoren 58, 963
- async() 575
- at() 91, 96, 859, 862, 867, 878, 887, 933
- atan() 790, 953
- atan2() 953
- atanh() 790
- ate 446
- atexit() 955
- atof(), atoi(), atol() 955
- Atom-Uhr 618
- atomic 572
- atomic_ref 574
- Attribut
 - Compilersteuerung 965
 - einer Klasse 177, 978
- Aufforderung 978
- Aufzählungstyp 85
- Ausdruck
 - Auswertung 58, 59
 - Definition 39
 - mathematischer 51, 131
- Ausgabe 104, 107, 427
 - benutzerdefinierter Typen 428
 - Datei- 108, 423
 - Formatierung
 - mit Flags 433
 - mit std::format() 429

- mit Manipulatoren 437
- Weite der 431, 433
- Ausgabeoperator 371, 427
- Ausnahme 349, 978
- Auswertungsreihenfolge 51, 59, 964
- auto 44, 97, 100, 468, 692
 - als Rückgabetyt 159, 182, 318
- auto(), auto{} 809
- auto[] 102, 837
- automatische Variable 150
 - make 655
- Autotools (GNU) 671

B

- back() 859, 862, 864, 867, 870, 933
- back_inserter(), _insert_iterator 901
- Backslash
 - Zeichenkonstante \ 55
 - Zeilenfortsetzung 140
- Backspace 55
- bad() 445
- bad_alloc 355
- badbit 443
- bad_cast 340, 355
- bad_function_call 355
- bad_typeid 343, 355
- bad_weak_ptr 355
- base() 899
- basefield 435
- basic_string 931
- basic-Streamklassen 424
- Basisklasse 302
 - virtuelle B. 336
 - Konstruktor 334
 - Subobjekt 338
 - und virtueller Destruktor 326
- Bedingungsausdruck 67
- Bedingungsoperator ?: 70
- beg 447
- begin()
 - Container 465, 854
 - Namespace std 226, 855, 898
 - string 275, 932
 - vector 226
- Belegungsgrad 883
- benutzerdefinierte
 - Datentypen 85, 88
 - Klassen 371
 - Literale 412
 - Typen (Ausgabe) 428
 - Typen (Eingabe) 449
- Benutzungszählung 946
- Bereichsnotation 856, 980
- Bereichsoperator :: 62, 180, 315
 - namespace 152
- Bibliothek
 - C 167, 951
 - C++ 827
- Bibliotheksmodul 146
 - dynamisch 668
 - statisch 666
- Bidirectional-Iterator 893
- Big Three (Regel) 256
- Big Three/Five/Zero (Regel) 496
- binary 111, 446
- binary_search() 774
- bind 845
- Binden 146
 - dynamisches *siehe* dynamisches B.
 - statisches *siehe* statisches Binden
- Binärdatei 260
- binäre Ein-/Ausgabe 257
- binärer Operator 369
 - optimiert 493
- binäres Prädikat 748, 788, 906
- binäre Zahlendarstellung 46
- Binärzahl 42
- Bit
 - feld 103
 - Operatoren 45, 46
 - pro Zahl 40
 - Verschiebung 46
- bit_cast 104
- bitset 839
- bitweises ODER, UND, XOR 45
- Block 30, 32, 61, 65, 209
 - und dyn. Objekte 237
- bool 57
- boolalpha 57, 434, 438
- break 71, 82
- bsearch() 955
- Bstatic (Makefile) 667

- Bucket 883
- bucket() 885
- Byte 54
 - std::byte 87, 954
 - Reihenfolge 618
- C**
- C++ Core Guidelines 113
- C++-Schlüsselwörter 961
- call wrapper 557
- Callback-Funktion 262, 846
- canonical() (Filesystem) 818
- capacity() 860, 934
- capturing group 536
- C-Array 222
 - mehrdimensionales 246
- case 71
 - lokale Variablen 74
- <cassert> 143, 951
- cast *siehe* Typumwandlung
- catch 350
- cbegin()
 - Container 854
 - Namespace std 855, 898
 - string 933
- <cctype> 804, 952
- cdecl 269
- ceil() 953
- cend()
 - Container 854
 - Namespace std 855, 898
 - string 933
- cerr 105, 424
- C-Funktionen einbinden 169
- char 54
- char* 227
- char* const vs. const* char 221
- char_traits 228, 281, 529
- char8_t, char16_t, char32_t 411, 916
- ChatGPT 635
- C-Header 951
- <chrono> 385, 548
- cin 32, 105, 424, 444
- clamp() 812
- class 178, 308
- class (bei Template-Parametern) 161
- clear() 445, 858, 874, 936
- Client-Server
 - Beziehung 179
 - und callback 262
- clock(), clock_t 958
- clog 105, 424
- close() 108
- CMake 671
- <cmath> 51, 951, 953
- co_await 583
- code bloat 298
- Code-Formatierung 113
- collate 920
- combine() 914
- <compare> 399
- compare() 921, 938
- Compilationsmodell 299
- Compiler 29, 30, 33, 145, 177, 179
 - befehle 962
 - direktiven 137
 - und Templates 164
 - Typumwandlung 201
- <complex> 789
- Computerarithmetik 51
- Concepts 530
- conditional 529
- configure 671
- conj() 790
- connect() 588
- const 53
 - correctness 181
 - Elementfunktionen 178, 180
 - globale Konstante 151
- const& *siehe* Referenz auf const
- const char* vs. char* const 221
- consteval 159
- constexpr
 - Funktion 156
 - Konstante 53
 - Konstruktor/Methode 195
- constinit 292
- const_iterator 853
- const_local_iterator 885
- const_pointer 853
- const_reference 853
- const_reverse_iterator 855, 900

constraint, Vererben von 331
Container 458
 implizite Datentypen 853
 Methoden 854
Container-Adapter 868
container_type 868
contains()
 Container 874
 string 938
Contiguous-Iterator 893
continue 82
convertible_to 534
copy semantics 484
copy() (Filesystem) 820
copy() 800, 933
copy_backward() 800
copy_if() 802
copy_n() 803
copy_constructible 534
copy_options 821
copy_result (Ranges) 734
copysign() 953
co_return 581
Coroutinen 579
cos(), cosh() 789, 953
count() 842, 874
 Algorithmus 750
count_if() 750
cout 32, 105, 424
co_yield 580
__cplusplus 169
crbegin()
 Container 855
 Namespace std 855, 898
 string 933
create_directory() 821
crend()
 Container 855
 Namespace std 855, 898
 string 933
critical section 560
CRLF 621
<cstdlib> 40, 220, 225, 954
<cstdliblib> 130, 261
c_str() 933
C-String 227

 Länge 230, 231
<cstring> 227, 263, 273, 276, 955
<ctime> 957
ctype 721, 722, 920, 921
cur 447
curr_symbol 925
current_path 818
<cwctype> 952
CXXFLAGS 654

D

DALL·E 2 635
dangling
 pointer 237
 ranges 735
 reference 482
data race 560, 562
data() 859, 862, 933
Datagramm 616
Datei
 ASCII 258
 binär 260
 Ein-/Ausgabe 108, 423
 kopieren 111, 820
 löschen 818
 Öffnungsarten 446
 öffnen 111
 Positionierung 447
 schließen 109
 umbenennen 822
 Zugriffsrechte 815
Daten
 als Attributwerte 979
 static-Element- 286
Datenbankanbindung 639
Datenkapselung 979
Datentypen 33, 39, 175
 abstrakte *siehe* abstrakter
 Datentyp
 benutzerdefinierte 85
 int und unsigned 60, 70
 logische 57
 parametrisierte 161, 293
 polymorphe 323
 strukturierte 88
 zusammengesetzte 85

- Datum
 - Klasse 386
 - regulärer Ausdruck 791
 - days 386
 - daytime (Port 13) 611
 - Deadlock 579
 - dec 434, 438
 - decay, decay_t 528
 - decimal_point() 923, 924
 - decltype 283
 - decltype(auto) 285, 525, 834
 - declval 286
 - default (in switch-Anweisung) 71
 - = default 191
 - default constructor 182
 - default_delete<X[]> 698
 - defaultfloat 434
 - Default-Parameter *siehe* vorgegebene P.
 - #define 137, 140
 - Definition 148, 979
 - von static-Elementdaten 287
 - von Objekten 185
 - Deklaration 32, 33, 148, 979
 - einer Funktion 117
 - Funktionszeiger 260
 - Lesen einer D. 267
 - in for-Schleifen 79
 - Deklarationsanweisung 64
 - Dekrementierung 45, 46
 - Dekrementoperator 388
 - Delegation 345
 - delegierender Konstruktor 194
 - delete 234, 236, 289, 324, 326
 - überladen 404
 - delete [] 238
 - = delete 191, 694
 - Deleter 699
 - <deque> 852, 867
 - deque 867
 - Dereferenzierung 218, 262
 - derived_from 534
 - Destruktor 208, 335, 496
 - implizite Deklaration 208
 - und exit() 210
 - virtueller 324
 - detach() 553
 - Dezimalpunkt 48, 923
 - Dialog 599
 - diamond problem 338
 - difference_type 853
 - Differenz (Menge) 780
 - difftime() 958
 - digits, digits10 810
 - distance() 897
 - Distribution 657, 979
 - div(), div_t 954
 - divides 844
 - Division durch 0 356
 - DNS 609
 - do while 76
 - domain_error 355
 - double 48, 224
 - nicht als Laufvariable 80
 - korrekter Vergleich 690
 - downcast 340
 - Drei (die großen Drei) 256, 496
 - Drei-Wege-Vergleich *siehe* Spaceship-Operator
 - drop() 471
 - Dubletten entfernen 746
 - Durchschnitt (Menge) 779
 - DYLD_LIBRARY_PATH 669
 - dynamic_cast<>() 340
 - dynamic_pointer_cast 948
 - dynamischer Typ 342
 - dynamisches
 - Array 374
 - Binden 260, 315, 980
 - Datenobjekt 233
- ## E
- e, E 48
 - Editor 29
 - egrep 535
 - Ein- und Ausgabe 104
 - Einbinden von C-Funktionen 169
 - Eingabe 425
 - Datei- 108, 423
 - von Strings 106
 - benutzerdefinierter Typen 449
 - Einschränkung *siehe* constraint
 - Elementdaten, Zeiger auf 266
 - Elementfunktion 176

- als Funktionsobjekt 848
 - spezielle 496
 - Zeiger auf 265
 - Ellipse
 - in catch-Klausel 351
 - template parameter pack 514
 - else 66
 - `#else`, `#elif` 137
 - `emplace()` 857, 869, 870, 872
 - `emplace_back()` 558, 859, 865, 867
 - `emplace_front()` 864, 867
 - `empty()` 854, 869–871, 933
 - Namespace `std` 855
 - `enable_if` 528
 - `end` 447
 - `end()`
 - Container 465, 854
 - Namespace `std` 226, 855, 898
 - `string` 275, 932
 - `vector` 226
 - `#endif` 139
 - `endl` 55, 438
 - oder `'\n'`? 107
 - `ends` 438
 - `ends_with()` 938
 - ENTER** 31, 105
 - »enthält«-Beziehung 336
 - `enum` 85
 - Environment, `env[]` 243
 - `EOF` 426
 - `eof()` 351, 426, 427, 445, 460
 - `eofbit` 443
 - `epsilon()` 810
 - `equal()` 788
 - `equal_range()` 776, 874
 - `equalsIgnoreCase()` 722
 - `equal_to` 845
 - `erase()` 858, 874, 936
 - ereignisgesteuerte Programmierung 586
 - `errc` 717
 - `errno` 348
 - `#error` 144
 - `error_code` 816
 - Escape-Sequenz 55
 - Exception 978
 - arithmetische Fehler 356
 - und Destruktor 349
 - Handling 349
 - Hierarchie 353
 - Speicherleck durch `Exc.` 696
 - `<exception>` 355
 - `exception` 353
 - Exception-Sicherheit 359
 - `exchange()` 804
 - `exclusive_scan()` 741
 - ExecutionPolicy 814
 - `exists()` (Filesystem) 819
 - `exit()` 130, 955
 - und Destruktor 210
 - Exklusiv-Oder (Menge) 781
 - `exp()` 790, 953
 - `expected` 361
 - `explicit` 191
 - explizite Instanziierung von Templates 300
 - Exponent 48–50
 - `export` 170
 - `extension()` (Filesystem) 818
 - `extent` 527
 - `extern` 149–151
 - `extern "C"` 169
 - `extern template` 299
 - `external linkage` 154
 - `extract()` 875
- ## F
- `f`, `F` (Suffix) 48
 - `fabs()` 953
 - Facette 920
 - `fail()` 445
 - `failbit` 443, 451
 - `fakultaet()` 116
 - `[[fallthrough]]` 73
 - Fallunterscheidung 71
 - `false` 57
 - `false_type` 522
 - `falsename()` 923
 - Fehlerbehandlung 347
 - Ein- und Ausgabe 443
 - Fibonacci 512, 743
 - `__FILE__` 145
 - `filename()` 818

- Filesystem
 - canonical() 818
 - copy() 820
 - create_directory() 821
 - current_path() 818
 - exists() 819
 - extension() 818
 - filename() 818
 - is_directory() 819
 - recursive_directory_iterator 824
 - remove() 818
 - remove_all() 819
 - stem() 818
- filesystem 108
- fill() 433, 735
- fill_n() 735
- Filter, filter() 470
- final 327, 695
- find()
 - Algorithmus 767
 - assoziative Container 874
 - string 937
- find_end() 771
- find_first_of() 768
- find...-Methoden (string) 937
- fixed 434, 436, 438
- Fixture 677
- flache Kopie 254
- flags() 434
- flip() 861
 - Bitset 841
- float 48
- float16_t und weitere float-Typen 52
- floatfield 435
- floating_point 534
- floor() 953
- flush 428
- flush() 107
- flush (Manipulator) 438
- fmod() 953
- fmtflags 434
- Fold-Expression 516
- for 78
 - Kurzform 99
- foreach(), for_each_n() 799
- Formalparameter 118
- Formatierung 429
- forward() 834
- Forward-Iterator 892
- <forward_list> 852
- forwarding reference 492
- frac_digits() 925
- Fragmentierung (Speicher) 238
- Framework 587
- free store 234
- free() 407
- frexp() 953
- friend 279
- from_chars() 717
- front() 859, 862, 864, 867, 870, 933
- front_inserter(), _insert_iterator 901
- <fstream> 108
- fstream 424, 447
- Füllzeichen 430, 433
- __func__ 145
- function 846
- <functional> 355, 844
- Funktion 116
 - frei oder global 153
 - mit Gedächtnis (static) 120
 - mit initializer_list 856
 - klassenspezifische 287
 - mathematische 51, 953
 - vorgegebene Parameterwerte 127
 - Parameterübergabe
 - per Referenz 124
 - per Wert 122
 - per Zeiger 239
 - rein virtuelle 319
 - mit Definition 320
 - static 287
 - alternative Syntax 283
 - Überschreiben 314
 - virtuelle *siehe* virtuelle Funktionen
- Funktionsobjekte 396, 440
 - function 846
 - mem_fn 848
- Funktions-Template 161
- Funktork *siehe* Funktionsobjekte
- <future> 355
- future 577

Fünf (die großen Fünf) 496

G

Ganzzahlen 40
garbage collection 238
gcd() 203, 813
gegenseitige Abhängigkeit von
 Klassen 215
Genauigkeit 49
Generalisierung 302
generate(), generate_n() 736
generische Programmierung 458
GET (http) 621
get() 105, 258, 425, 426
getenv() 955
getline() für Strings 107, 939
getline(char*,...) 426
getloc() 925
get_money() (Manipulator) 438
ggT 75
 std::gcd() 813
 schnell 203
Gleichheitsoperator 372
 bei Vererbung 420
Gleichverteilung 795
Gleitkommazahl 53
 Syntax 48
global 62
 Funktion 153
 Namensraum 167
 Variable 149, 150
glvalue 479
gmtime() 958
GNU Autotools 671
good() 445
goodbit 443
goto 84
grafische Benutzungsschnittstelle 585
greater, greater_equal 845
greedy (regex-Auswertung) 538
Grenzwerte von Zahltypen 809
grouping() 923, 924
Groß- und Kleinschreibung 31
größter gemeinsamer Teiler *siehe* ggT
guard (Threads) 561
Gültigkeitsbereich 130

Block 61
Datei 154
Funktion 119
Klassen 178
 und new 237

H

hängender Zeiger 237
hardware_concurrency() 549
has_facet() 914
hash() 921
hasher 885
Hash-Funktion 883, 884
has_infinity 810
__has_include 140
has_sort_function 522
has_value() (optional) 362
»hat«-Beziehung 712
Header 33, 167
 Datei 145
 Inhalt 149
 Http 621
 der Standardbibliothek 828
Heap 782
hex 434, 438
Hexadezimalzahl 42
hexfloat 434
Host Byte Order 618
hours 548
hängende Referenz 482

I

-I Compileroption 137
iconv 919
IDE 36
identity 734
Identität von Objekten 175, 980
IEC 60559, IEEE 754 49
if 66
if consteval 158
if constexpr 142, 516
#if, #ifdef, #ifndef 137
ifstream 108, 424
ignore() 426
imag() 789, 790
imbue() 721, 912

- Implementation 145
 - sdatei, Inhalt 149
 - svererbung 343
- implizite Deklaration
 - Destruktor 208
 - Konstruktor 182
 - Zuweisungsoperator 379, 419
- import 170
- in 446
- #include 32, 137
- Include-Guard 138
- includes() 778
- inclusive_scan() 740
- Indexoperator 90, 222, 225, 250, 377
 - mehrdimensionaler 417
- index_sequence 842
- infinity() 810
- Initialisierung
 - array 862
 - direkte I. der Attribute 184
 - C-Array 224, 247
 - einfacher Datentypen 43
 - mit Element-Initialisierungsliste 183
 - Konstante in Objekten 183, 288
 - globaler Konstanten 149, 151
 - Reihenfolge 292
 - mit {}-Liste 192, 856, 898
 - mit konstruktor-interner Liste 288
 - und move() 490
 - von Objekten 182
 - von Referenzen 970
 - Reihenfolge der Initialisierung
 - von Attributen 183
 - von Funktionsargumenten 136
 - in for-Schleife 78
 - von static-Elementdaten 287
 - und virtuelle Basisklassen 338
 - und Vererbung 306
 - und Zuweisung 44, 187
- initializer_list 193, 377, 811, 856
 - für zweidimensionales Array 856
 - und for-Schleifen 194
- Inklusionsmodell 299
- Inkrementierung 45, 46
- Inkrementoperator 385
- inline
 - Elementfunktion 181
 - Funktion 155
 - Konstante 151, 290
 - Variable 155, 290
- inner_product() 739
- innere Klasse 466, 641
- inplace_merge() 766
- Input-Iterator 892
- insert() 858, 874, 882, 935
- inserter() 902
- insert_iterator 902
- insert_or_assign 878, 887
- Instanz 174, 980
- Instanziierung von Templates 295
 - explizite 300
 - ökonomische (bei vielen Dateien) 298
- int 32, 40
- int-Parameter in Templates 296
- intX_t, int_fastX_t, int_leastX_t
 - (X = 8, 16, 32, 64), intmax_t 47
- integer_sequence 842
- integral 534
- integral_constant 522
- integral promotion 60
- Interface (UML) 706
- internal 434, 438
- internal linkage 154
- Internet-Anbindung 607
- Intervall (und Notation) 980
- invalid_argument 355
- <iomanip> 438, 439
- ios 424, 443, 445
 - failure 445
 - Flags zur Dateipositionierung 447
 - Methoden 434, 445
- <ios> 438
- ios_base 424
 - binary 111
 - Fehlerstatusbits 443
 - Flags 434, 435
 - Manipulatoren 438
- iostate 443
- <iostream> 33, 424, 425, 428, 438
- iota, iota_view 737

- IPv4, IPv6 609
 - IPv4-Adresse (regulärer Ausdruck) 793
 - is() 921
 - is_abstract 527
 - isalnum(), isalpha() 920, 952
 - is_arithmetic 525, 526, 531
 - is_array 526
 - is_base_of 527
 - isblank() 952
 - is_bounded 810
 - is_class 521, 526
 - isctrl() 920, 952
 - is_const 527
 - is_convertible 527
 - isdigit() 135, 920, 952
 - is_directory() (Filesystem) 819
 - is_enum 526
 - is_exact 810
 - is_final 527
 - is_floating_point 526
 - is_function 526
 - is_fundamental 527
 - isgraph() 920, 952
 - is_heap(), is_heap_until() 786
 - is_iec559, is_integer 810
 - is_integral 526
 - islower() 920, 952
 - is_lvalue_reference 526
 - is_modulo 810
 - isnan() 400
 - is_null_pointer 526
 - ISO 10646 916
 - ISO 8859-1, ISO 8859-15 915
 - is_partitioned() 758
 - is_permutation() 754
 - is_pointer 526
 - is_polymorphic 527
 - isprint() 920, 952
 - ispunct() 920
 - is_reference 526
 - is_rvalue_reference 526
 - is_same 527
 - is_signed 527, 810
 - is_sorted(), is_sorted_until() 760
 - isspace() 920, 952
 - Ist-ein-Beziehung 302, 311, 330
 - istream 424, 425
 - Istream-Iterator 902
 - istream::seekg(), tellg() 447
 - istream::ws 438
 - istreamstring 424, 448
 - is_unsigned 527
 - isupper() 920, 952
 - is_void 526
 - isxdigit() 920, 952
 - iter_swap() 803
 - Iterator 226, 275, 459, 463, 891
 - Adapter 899
 - Bidirectional 893
 - Contiguous 893
 - Forward 892
 - Input 892
 - Insert 900
 - Output 892
 - Random Access 893
 - Reverse 899
 - Stream 902
 - Tag 893
 - Zustand 464
 - <iterator> 891
 - iterator 853
 - iterator_category 892
- ## J
- Jahr 957
 - join() 552, 553
 - jthread (Klasse) 554
- ## K
- Kardinalität 709
 - Kategorie (locale) 920
 - key_equal 885
 - key_type 873, 880
 - key_comp(), key_compare 875
 - KI 635
 - Klammerregeln 58
 - Klasse 174, 177, 981
 - abgeleitete *siehe* abgeleitete Klasse
 - abstrakte 319
 - Basis- *siehe* Basisklasse
 - Deklaration 179

- innere 466, 641
 - konkrete 320
 - Ober- *siehe* Oberklasse
 - für einen Ort 178
 - Unter- *siehe* Unterklasse
 - für rationale Zahlen 199
 - Klassenname (typeid) 343
 - klassenspezifische
 - Daten und Funktionen 286
 - Konstante 290
 - Klassen-Template 293
 - Klassifikation 302, 981
 - Kleinschreibung 31
 - kleinstes gemeinsames Vielfaches
 - siehe* lcm()
 - Kollisionsbehandlung 883
 - Kommandointerpreter 653
 - Kommandozeilenparameter 243
 - Kommaoperator 59, 81, 519
 - Kommentar 30
 - komplexe Zahlen 789
 - Komplexität 986
 - Komposition 712
 - konkrete Klasse 320
 - Konsole
 - auf UTF-8 einstellen 915
 - Konstante 52
 - globale 149, 151
 - klassenspezifische 290
 - konstante Objekte 180
 - Konstruktor 179, 182
 - allgemeiner 184
 - implizite Deklaration 182
 - delegierender 194
 - erben 312
 - Kopier- *siehe* Kopierkonstruktor
 - vorgegebene Parameterwerte 185
 - Typumwandlungs- *siehe*
 - Typumwandlungskonstruktor
 - Kontrollstrukturen 64
 - Konvertieren von Datentypen *siehe*
 - Typumwandlung
 - Kopie, flache/tiefe 254
 - Kopieren
 - von Dateien 111
 - von Zeichenketten 232
 - Kopierkonstruktor 187, 377, 496
 - Auslassung durch Compiler 189
 - Kreuzreferenzliste 731
 - kritischer Bereich 560
 - Kurzform-Operatoren 45
- ## L
- l, L, ll, LL (Suffix) 42
 - Lambda-Funktionen 501
 - LANG 912
 - late binding 260
 - Laufvariable 78, 79
 - Laufzeit 218
 - und Funktionszeiger 260
 - und new 234
 - und Polymorphie 315
 - Typinformation 342
 - lcm() 813
 - LD_LIBRARY_PATH 669
 - ldd 669
 - ldexp() 953
 - ldiv(), ldiv_t 954
 - left 434, 438
 - length() (C-String, constexpr) 228, 281, 529
 - length() (string) 933
 - length_error 355
 - less, less_equal 845
 - lexicographical_compare() 755
 - lexicographical_compare_three_way() 756
 - lexikografischer Vergleich 401, 981
 - <limits> 41, 48, 809
 - __LINE__ 145
 - Linken 150, 167
 - dynamisches 668, 981
 - internes, externes 154
 - statisches 666, 981
 - Linker 35
 - linksassoziativ 58, 246
 - list 864
 - <list> 832, 852
 - Liste
 - Initialisierungs- 183, 288
 - Initialisierungs- (bei C-Arrays) 247
 - Liste (Klasse) 465

Literal 228, 981
 benutzerdefiniert 412
 String 411
 Zahl- 53
 Zeichen- 54, 916
load_factor() 885
local_iterator 885
<locale> 911
localtime() 958
log(), log10() 789, 953
logic_error 354, 355
logical_and, _not, _or 845
logischer Datentyp 57
logische Negation 57
lokal (Block) 61
lokale Objekte 221
long 40
long double 48
lower_bound() 775, 875
lvalue 480
L-Wert 480
Länge eines Vektors 739

M

magic number 982
main() 30, 32, 130
MAKE 664
make 651
 automatische Ermittlung von
 Abhängigkeiten 657
 parallelisieren 665
 rekursiv 664
 Variable 654
Makefile 146, 652
make_from_tuple() 838
make_heap() 784
make_index_sequence 842
make_integer_sequence 842
make_pair() 836
make_shared 697, 947
make_tuple() 837
make_unique 270, 697, 945
Makro 140
malloc() 407
Manipulatoren 437
Mantisse 49

<map> 852, 876
mapped_type 878
match_results 542
mathematische Funktionen 953
mathematischer Ausdruck 51
Matrix
 C-Array 246
 Klasse 251
 operator[] 415
max() 810
max(initializer_list<T>) 811
max_bucket_count() 885
max_element() 743
max_exponent, max_exponent10 810
max_load_factor() 886
max_size() 854
[[maybe_unused]] 965
mehrdimensionaler Indexoperator 417
mehrdimensionales C-Array 246
Mehrfachvererbung 304, 333, 335
MeinString (Klasse) 273
mem_fn() 848
member function *siehe* Element-
 funktion
memcpy() 376
<memory> 355, 832
memory leak 237, 696
Mengenoperationen auf sortierten
 Strukturen 777
merge() 765, 865, 874
mergesort() 766
messages 920, 927
Metaprogrammierung 509
Methode 174, 176, 982
 Regeln zur Konstruktion
 von Prototypen 688
microseconds 549
midpoint() 813
milliseconds 549
MIME 982
min() 810
min(initializer_list<T>) 811
min_element() 743
min_exponent, min_exponent10 810
minmax(...) 811
minmax_element() 743

- minus 844
- Minute 957
- minutes 548
- mischen 765
- mismatch() 786
- mkdir 659
- mktime() 958
- modf() 953
- modulare Programmgestaltung 145
- Module 169
- Modulo 45
- modulus 844
- monadische Operationen 364
- Monat 957
- monetary 920, 924
- money_get 925
- money_punct 924
- money_put 926
- Monitor-Konzept 570
- move() 489
 - Initialisierung 490
- move semantics 484
- move(), move_backward()
 - Container-Bereich 834
- move_constructible 534
- moving constructor 487
- mt19937 746, 794
- multimap 880
- multiplies 844
- Multiplikationsoperator 383
- Multiplizität (UML) 709
- multiset 882
- mutable
 - Attribut 180
 - Lambda-Funktion 506
- mutex 560
- N**
- '\n' 33
 - oder endl? 107
 - und regex_replace 544
- Nachbedingung 136, 982
- Nachkommastellen 48
 - precision 436
- Name 39
 - einer Klasse (typeid) 343
- name() 343, 914
- Named Return Value Optimization (NRVO) 189
- Namenskonflikte bei Mehrfach-
vererbung 335
- Namespace 63, 151
 - anonym 154
 - in Header-Dateien 154
 - Verzeichnisstruktur 661
- namespace 151
- namespace std 63
- NaN (not a number) 400
- nanoseconds 548
- narrow() 922
- ationale Sprachumgebung 912
- NDEBUG 143
- negate 844
- Negation
 - bitweise 45, 47
 - logische 57, 58
- negative_sign() 925
- neg_format() 925
- Network Byte Order 618
- Netzwerkprogrammierung 607
- neue Zeile 55, 66
- new 233, 236, 289
 - Fehlerbehandlung 358
 - Placement-Form 949
 - überladen 404
- <new> 355, 949
- new_handler 358
- next() 897
- next_permutation() 753
- noboolalpha 438
- Nodehandle 875
- [[nodiscard]] 118, 202
- noexcept 352
- none() (Bitset) 842
- none_of 751
- norm() 789, 790
- Normalverteilung 797
- noshowbase, -point, -pos 438
- noskipws 438
- not_equal_to 845
- nothrow 359
- notify_one(), -_all() 566, 703

nounitbuf, nouppercase 438
now 386, 549
npos 932
nth_element() 764
NTP – Network Time Protocol 618
NULL 220, 954
nullopt 361
nullptr 220
Null-Zeiger und new 359
<numbers> 196
numeric 920, 923
numeric_limits 48, 809
numerische Auslöschung 50
numerische Umwandlung 715, 718, 940
num_get, num_put 923
NummeriertesObjekt (Klasse) 287
numpunct 923

O

O-Notation 986
Oberklasse 302, 314, 982
 Subobjekt einer 306
 Subtyp einer 310
 Zugriffsrechte vererben 308
Oberklassenkonstruktor 303, 306
object slicing 311, 319
Objekt 28, 175, 179, 983
 -code 35
 dynamisches 233
 als Funktions- 396
 -hierarchie 336
 Identität *siehe* Identität von
 Objekten
 Initialisierung 182
 konstantes 180
 -orientierung 173
 Übergabe per Wert 188
 verkettete Objekte 236
 verwitwetes 237
 vollständiges 338, 986
oct 434, 438
ODER
 bitweises 45
 logisches 58
Öffnungsarten für Streams 446
offsetof 954
ofstream 108, 424
Oktalzahl 42
omanip 439
one definition rule 149
open() 108
Open Source 983
Operator
 arithmetischer 45
 binärer 369
 Bit- 45
 für char 57
 als Funktion 368
 Kurzform 46
 für Literale 410
 für logische Datentypen 57
 Präzedenz 58, 963
 relationale 45, 57
 Syntax 368
 Typumwandlungs- 389
 unärer 369
 für ganze Zahlen 45
operator delete() 404
operator new() 404
operator()() 396
operator*() 384, 390, 463
operator*=() 383
operator++() 385, 387, 463
operator++(int) 692
operator+=() 371
operator->() 390
operator<=>() 398
operator<<() 371, 427
operator=() 380
operator==() 463, 842
 bei Vererbung 420
operator==() 372
operator>>() 425
operator[]() 416, 859, 862, 867, 893
operator!=() 463
Optimierung
 Compileroption 963
 Vermeiden temporärer Objekte 189
optional 361
or_else() 364
Ort (Klasse) 178
ostream 371, 424, 427

- ostream::endl, ends, flush() 438
 - Ostream-Iterator 902
 - ostream::seekp(), tellp() 447
 - ostringstream 424, 448
 - osyncstream 568
 - out 446
 - out_of_range 355
 - Output-Iterator 892
 - overflow 43, 50
 - overflow_error 355
 - override 317, 694
- P**
- packaged_task 577
 - pair 835
 - Parameter
 - expansion 515
 - einer Funktion 117
 - Pack 514
 - übergabe
 - per Referenz 124
 - per Wert 122
 - per Zeiger 239
 - parametrisierte Datentypen 161, 293
 - »part-of«-Beziehung 712
 - partial_ordering 400
 - partial_sort(), partial_sort_copy 762
 - partial_sum() 740
 - partielle Spezialisierung von
 - Templates 892
 - partition() 757
 - partition_copy(), partition_point() 758
 - path 108, 815
 - patsubst 657
 - peek() 427
 - perfect forwarding 834
 - Performance 475
 - Permutationen 752
 - Pfad 108
 - Pfeiloperator 221
 - PHONY 654
 - Pipe (ranges) 470
 - Placement new/delete 949
 - plus 844
 - pointer 853
 - Pointer, smarte 390
 - polar() 790
 - polymorpher Typ 323, 342
 - Polymorphismus 315, 983
 - pop() 869, 870, 872
 - pop_back() 860, 865, 868
 - pop_front() 865, 867
 - pop_heap() 783
 - portabel (Zeichensatz) 918
 - pos_format() 925
 - Positionierung innerhalb einer Datei 447
 - positive_sign() 925
 - POSIX 717, 912
 - POST (http) 626
 - postcondition *siehe* Nachbedingung
 - Postfix-Operator 385
 - pos_type 447
 - pow() 790
 - precision() 435, 436
 - precondition *siehe* Vorbedingung
 - prev() 897
 - prev_permutation() 752
 - PRINT (Makro) 142
 - printf() 513
 - Priority-Queue 871
 - private 178, 307
 - private Vererbung 343
 - Programm
 - ausführbares 35
 - Strukturierung 115
 - Programmierrichtlinien 113, 696
 - Projection (Ranges) 734
 - proj() 790
 - Projekt 36, 146
 - promise 577
 - protected 307
 - protected-Vererbung 345
 - Prototyp
 - Funktions- 116
 - einer Methode 178
 - Regeln zur Konstruktion 688
 - prvalue 480
 - Prädikat
 - Algorithmus mit P. 906
 - binäres 748, 788, 906
 - unäres 750
 - Präfix-Operator 385

Präprozessor 33, 136, 657
Präzedenz von Operatoren 58, 963
ptrdiff_t 40, 225, 855, 954
public 178, 307
Pufferung (Ein-/Ausgabe) 105, 428
push() 869, 870, 872
push_back() 860, 865, 867
 vector 94
push_front() 865, 867
push_heap() 784
put() 108, 258, 428
putback() 426
put_money() (Manipulator) 438

Q

qsort() 261, 955
Qt 587
QThread 602
Quantifizierer 538
Quellcode 35
Queue 870
<queue> 832, 852, 870
quoted() 438

R

race condition 560, 578
radix 810
RAII 561, 677, 697, 983
<random> 746
random access 464
Random-Access-Iterator 893
random_access_iterator 534
random_device 795
range based for 99
range_error 355
Ranges (Bereiche), <ranges> 469
rank 527
<ratio> 848
Rationale Zahl
 Klasse 199
 Template std::ratio 848
rbegin()
 Container 855, 899
 Namespace std 855, 898
 string 933
rdstate() 445

read() 257
real() 789, 790
Rechengenauigkeit 49, 80
rechtsassoziativ 58, 246
recursive_directory_iterator 824
reelle Zahlen 48
ref(), reference_wrapper 557, 850
reference
 bitset 840
 Container 853
 vector<bool> 860
reference collapsing rules 491
reference counting 946
Referenz 124, 133
 auf Basisklasse 318
 auf const 126, 187
 auf istream 425, 449
 auf Oberklasse 310, 318
 auf ostream 372, 427
 Parameterübergabe per 124
 auf R-Wert 484
 Rückgabe per 378
 -semantik 255, 475
 weiterleitende 492
Referenz-Qualifizierer 482
Regel der 0/3/5 *siehe* Rule of 0/3/5
<regex> 355, 543
regex_iterator 542
regex_match() 543
regex_replace() 544, 726
regex_search() 544, 724
reguläre Ausdrücke 535
Reihenfolge
 Auswertungs- 51, 59
 der Initialisierung
 von Attributen 183
 von Funktionsargumenten 136
 umdrehen 749
rein virtuelle Funktion 319
 mit Definition 320
reinterpret_cast<>() 225, 257
Rekursion 123
 Template-Metaprogrammierung 510,
 514
rekursiver Abstieg 130, 131
rekursiver Make-Aufruf 664

- relationale Operatoren 45, 57
 - remove()
 - Algorithmus 807
 - Datei/Verzeichnis 818
 - Liste 865
 - remove_all() Dateien/Verzeichnisse 819
 - remove_const 528
 - remove_if() 807, 865
 - remove_reference 527, 834
 - rename() Datei/Verzeichnis 822
 - rend()
 - Container 855, 899
 - Namespace std 855, 898
 - string 933
 - replace() 806, 936
 - replace_copy(), _if(), _copy_if() 806
 - requires 531
 - reserve() 860, 934
 - reset() (Bitset) 841
 - resetiosflags() 438
 - resize() 860, 865, 868, 934
 - resume() 580
 - return 33, 189
 - Return Value Optimization (RVO) 189
 - reverse() (list) 865
 - reverse(), reverse_copy() 749
 - reverse (Ranges) 471
 - reverse_iterator 855
 - Reverse-Iterator 899
 - reversible Container 855
 - right 434, 438
 - »rohes« Stringliteral 411
 - rotate(), rotate_copy() 744
 - round_error(), round_style() 810
 - RTTI 342
 - Rückgabotyp auto 159, 182, 318
 - Rule of zero/three/five 496
 - runtime_error 355
 - rvalue 480
 - R-Wert, Referenz auf R-Wert 480
- S**
- safe_iterator_t 735
 - safe_subrange_t 735
 - same_as 533
 - sample() 749
 - scan_is(), scan_not() 922
 - Schleifen 74
 - und Container 98
 - do while 76
 - for 78
 - und C-Strings 230
 - Tabellensuche 93
 - terminierung 76
 - while 74
 - Schlüsselwörter 961
 - Schnittmenge 779
 - Schnittstelle 145, 984
 - einer Funktion 121
 - Regeln zur Konstruktion 688
 - scientific 434, 436, 438
 - scope 61
 - scoped locking 561
 - scoped_lock 561
 - search() 770, 771
 - search_n() 773
 - seconds 548
 - seekg(), seekp() 447
 - Seiteneffekt 67, 117, 136, 231, 233
 - im Makro 143
 - Seitenvorschub 55
 - Sekunde 957
 - Selbstzuweisung 381, 498
 - Selektion 66
 - sentinel 93, 225
 - Sequenz
 - konstruktor 192, 193, 377
 - methoden (Container) 858
 - Server-Client
 - Beziehung 179
 - und callback 262
 - <set> 852, 880
 - set() (Bitset) 841
 - setbase() 438
 - set_difference() 780
 - setf() 434, 435
 - setfill() 438
 - set_intersection() 779
 - setiosflags() 438
 - setprecision() 438
 - setstate() 445
 - set_symmetric_difference() 781

- set_terminate() 356
- set_union() 779
- setw() 438
- SFINAE 523, 984
- shared_ptr 395, 697, 946
 - für C-Arrays 698
- short 40
- showbase, showpoint, showpos 434, 438
- showContainer() 524
- shrink_to_fit() 860, 868, 934
- shuffle() 746
- Sichtbarkeit 61
 - sbereich (namespace) 151
- sign() 953
- Signal 588, 590
- Signalton 55
- Signatur 128, 304, 316, 318, 984
- signed char 54
- signed_integral 534
- sin(), sinh() 789, 953
- single entry/ single exit 82
- size()
 - Container 854
 - Namespace std 90, 223, 855
 - string 96
 - vector 90
- size_t 40, 954
- size_type 853
- sizeof 224
- sizeof 40
- sizeof... (variadische Templates) 514
- Skalarprodukt 739
- skipws 434, 438
- sleep_for(), sleep_until() 548, 553
- Slot 588, 590
- Small String Optimization 689, 984
- Smart Pointer 390
 - und Exceptions 696
- Socket 611
- Sommerzeit 957
- Sonderzeichen 56
- sortable 533
- sort_heap() 785
- Sortieren
 - mit qsort() 262
 - mit sort() 760
 - stabiles 760
 - durch Verschmelzen 766
- source_location 144
- Spaceship-Operator 398, 854
- 852
- span 242, 889
- Speicher
 - klasse 150
 - leck 237, 696
 - platzfreigabe 220
- Spezialisierung
 - von Klassen 302
 - von Templates 163
- spezielle Elementfunktionen 496
- splICE() 865
- split() 714
- Sprachumgebung 912
- SQL 639
- sqrt() 51, 790, 953
- ssize() 99, 855
 - Namespace std 223
- <sstream> 448
- stable_partition() 757
- stable_sort() 761
- Stack 61
 - Klasse 293
- <stack> 832, 852, 868
- stack unwinding 349
- Standard
 - bibliothek
 - C 167, 951
 - C++ 827
 - header 168, 831
 - klassen 831
 - Typumwandlung 59, 264
 - Zeiger 264
- Standard-Ein-/Ausgabe 105
- starts_with() 938
- static 150
 - Attribute und Methoden 286
 - in Funktion 120
 - Initialisierungsreihenfolge 292
 - static und -Bstatic (Makefile) 667
- static_assert 143
- static_cast<>() 56, 340
- static operator() 398

- statisches Binden 315, 984
- Statusabfrage einer Datei 445
- std 63, 153
- <stdexcept> 355
- <stdfloat> 52
- stdio 434
- Stelligkeit (Template) 512
- stem() (Filesystem) 818
- Stichprobe 749
- STL 457
- stod() 716
- stoi() 716
 - und verwandte numerische
Konversionsfunktionen 940
- stop_requested(), stop_token() 555
- strcat(), strchr(), strcmp() 956
- strncat(), strncmp() 957
- strpbrk(), strrchr(), strstr() 956
- strcpy() 956
- strcspn() 956
- Stream
 - Iterator 902
 - Öffnungsarten 446
- stream 423
- streamsize 425
- strerror() 717, 956
- Streuspeicherung 882
- strftime() 958
- String 95
 - in Zahl umwandeln 715
 - Klasse MeinString 273
 - Literal 411, 916
- string 95, 931
 - append() 934
 - assign() 935
 - at() 96, 933
 - back() 933
 - begin() 932
 - capacity() 934
 - cbegin(), cend(), crbegin(), crend() 933
 - clear() 936
 - compare() 938
 - copy() 933
 - c_str() 933
 - data() 933
 - empty() 933
 - end() 932
 - erase() 936
 - find() 937
 - find...-Methoden 937
 - front() 933
 - insert() 935
 - length() 96, 933
 - max_size() 933
 - operator+=() 934
 - rbegin(), rend() 933
 - replace() 936
 - reserve() 934
 - resize() 934
 - shrink_to_fit() 934
 - size() 96, 933
 - substr() 938
 - swap() 933
- <string> 832
- stringstream 718
- string_view 282, 940
- <string_view> 832
- strlen() 228, 956
- strncpy() 233, 957
- strong_ordering 400
- strtod(), strtol(), strtoul() 955
- strtok() 956
- struct 88, 308
- strukturierte Bindung 102
- Stunde 957
- Subobjekt 303, 310, 334
 - in virtuellen Basisklassen 337
 - verschiedene 336
- subrange 472
- Substitutionsprinzip 330
- substr() 938
- Subtyp 304, 310, 330, 985
- Suffix 42, 48, 97
- suspend_always 582
- swap() 380, 854, 933
 - Algorithmus 803
- swap-Trick 256
- swap_ranges() 803
- switch 71
- symmetrische Differenz (Menge) 781
- Synchronisation 559

- Syntax 53
- Syntaxdiagramm
 - ?: Bedingungsoperator 70
 - do while-Schleife 76
 - enum-Deklaration 85
 - for-Schleife 78
 - Funktions-
 - aufruf 118
 - definition 117
 - prototyp 117
 - Template 161
 - if-Anweisung 66
 - mathematischer Ausdruck 131
 - operator-Deklaration 368
 - struct-Definition 88
 - switch-Anweisung 71
 - Typumwandlungsoperator 389
 - while-Schleife 74
- system() 955
- system_clock 386, 548
- system_error 355, 445
- Szenario 211
- T**
- Tabellensuche 93
- Tabulator 55
- Tag (des Monats) 957
- Tag Dispatching (und Alternative) 894
- tan(), tanh() 789, 953
- target (make) 652
- Taschenrechnersimulation 130
- Tastaturabfrage 105
- TCP 608
- »Teil-Ganzes«-Beziehung 712
- tellg(), tellp() 447
- Template
 - Alias 268
 - für Funktionen 161
 - Instanziierung von T. 295
 - explizite 300
 - ökonomische (bei vielen Dateien) 298
 - int-Parameter 296
 - für Klassen 293
 - Metaprogrammierung 510
 - Method (Design-Muster) 328, 420
 - Spezialisierung 163
 - partielle 892
 - als template-Parameter 298
 - variable Parameterzahl 512
- temporäres Objekt (Vermeidung) 189
- terminate() 355
- terminate_handler 356
- test() (Bitset) 842
- Test Driven Development 684
- Test-Suite 675
- Textersetzung 140
- this 244
- this->, *this bei Zugriff auf Oberklasselement 384
- this_thread 552
- thousands_sep() 923, 924
- Thread 547
- thread (Klasse) 550
- ThreadGroup 558
- Thread-Sicherheit 578
- throw 350
- tiefeKopie 254
- time 920
- time() 958
- time_t 957
- tm 957
- to_array() 863
- tolower() 721, 920, 922, 952
- top() 869, 872
- to_string()
 - bitset 842
 - Zahlkonvertierung 940
- to_ulong() (Bitset) 841
- toupper() 721, 917, 920, 922, 952
- trailing return type 283
- traits 891
- traits::eof() 426
- transform()
 - Algorithmus 804
 - collate 921
 - monadische Funktion 364
 - Ranges 471
- transform_exclusive_scan() 741
- transform_inclusive_scan() 741
- tree (Programm) 660, 824

- Trennung von Schnittstellen und Implementation 146
 - true 57
 - truename() 923
 - true_type 522
 - trunc 446
 - try 350
 - try_emplace() 887
 - Tupel, <tuple> 837
 - Typ 985
 - polymorpher bzw. dynamischer Typ 342
 - type cast *siehe* Typumwandlung
 - typedef 267
 - typeid() 342
 - type_info 342
 - <typeinfo> 355
 - typename (bei Template-Parametern) 161
 - Type Traits 521
 - Typinformation 324
 - zur Laufzeit 342
 - Typumwandlung
 - cast 56, 220, 262
 - durch Compiler 201
 - dynamic_cast<>() 340
 - mit explicit 191
 - implizite 60, 191
 - mit Informationsverlust 129
 - skonstruktor 190, 201
 - soperator 389
 - ios 445
 - Standard- 59
 - Zeiger 264
 - static_cast<>() 340
- U**
- u, U (Suffix) 42
 - u8 916
 - u8string, u16string, u32string 411
 - UCS 916
 - UDP 608, 616
 - Überladen
 - von Funktionen 128
 - von Operatoren *siehe* operator
 - Überlauf 43, 50
 - Überschreiben
 - von Funktionen 314
 - Übersetzung 146
 - seinheit 148
 - uintX_t, uint_fastX_t, uint_leastX_t (X = 8, 16, 32, 64), uintmax_t 47
 - Umgebungsvariable 243
 - UML 705
 - Umleitung der Ausgabe auf Strings 448
 - UND
 - bitweises 45, 47
 - logisches 58
 - #undef 139
 - undefined behaviour 238
 - underflow 51
 - underflow_error 355
 - unsigned_integral 534
 - unexpected 363
 - Unicode 423, 915
 - uniform_int_distribution 795
 - uniform_real_distribution 796
 - union 104
 - unique() Sequenzen 865
 - unique(), -_copy() Algorithmen 746
 - unique_lock 561
 - unique_ptr 270, 395, 697, 943
 - für C-Arrays 699
 - Unit-Test 673
 - unitbuf 428, 434, 438
 - unordered_map 886
 - unordered_multimap 888
 - unordered_multiset 889
 - unordered_set 888
 - unsigned 40
 - unsigned char 54
 - Unterklasse 302, 985
 - unärer Operator 369
 - unäres Prädikat 750
 - upper_bound() 775, 875
 - uppercase 434, 438
 - URI, URL 608, 726
 - URL-Codierung 621
 - use case 211
 - use_facet() 721, 722, 914
 - using
 - enum 87
 - Deklaration 309, 344

- Namespace
 - Deklaration 152
 - Direktive 152
 - statt typedef 267
 - Template-Alias 268, 365
- UTC 958
- UTF 411, 985
- UTF-8 55, 915
- <utility> 286, 489, 803, 834, 837

V

- valarray 251
- valgrind 409
- value(), value_or() (optional) 362
- value_comp(), value_compare 875
- value_type 853, 873, 878
- Variable 33
 - automatische 150
 - globale 149, 150
 - make 654
 - Name 39
- variadic templates 512
- variant 843
- vector 859
 - at() 91
 - push_back() 94
 - size() 90
- <vector> 832, 852, 859
- vector<bool> 860
- Vektor 89
 - Klasse 374
 - Länge (geom.) 739
- Verbundanweisung 65
- verdecken (Methode) 304, 309
- Vereinigung (Menge) 779
- Vererbung 301, 985
 - der abstrakt-Eigenschaft 320
 - von constraints 331
 - der Implementierung 344
 - Mehrfach- 333
 - private 343
 - protected 345
 - von Zugriffsrechten 308
 - und Zuweisungsoperator 419
- Vergleich
 - von double-Werten 690

- bei Vererbung 420
- Verschiebung (*move*) 484
- verschmelzen (*merge*) 765
- Vertrag 331, 986
- verwitwetes Objekt 237
- Verzeichnis
 - anlegen 821
 - anzeigen 823
 - kopieren 820
 - löschen 818
 - umbenennen 822
- Verzeichnisbaum
 - anzeigen 824
 - make 661
- Verzweigung 66
- View
 - Definition 281
 - ranges 469
- virtual 316, 318, 326
- virtuelle Basisklasse 336
- virtuelle Funktionen 316, 318
 - private 328
 - rein- 319
- virtueller Destruktor 324
- void 220
 - als Funktionstyp 117
- void* 262
 - Typumwandlung nach 220
- volatile 986
- vollständiges Objekt 338, 986
- Vorbedingung 136, 986
- vorgegebene Parameterwerte
 - in Funktionen 127
 - in Konstruktoren 185
- Vorkommastellen 48
- Vorrangregeln 58, 963
- Vorwärtsdeklaration 215

W

- Wahrheitswert 57
 - zufällig erzeugen 798
- wait() 565
 - mit Lambda-Funktion 570
- #warning 144
- Warteschlange 870
- Wartung 113

- wchar_t 54, 916, 954
 - weak_ordering 401
 - weak_ptr 948
 - Webserver 627
 - Weite der Ausgabe 431, 433
 - weiterleitende Referenz 492
 - Wert
 - eines Attributs 978
 - Parameterübergabe per 122
 - Wertebereich ganzer Zahlen 42
 - Wertsemantik 255, 459, 475
 - Performanceproblem 477
 - what() 354
 - while 74, 230, 232
 - whitespace *siehe* Zwischenraum-
zeichen
 - wide character 54
 - widen() 922
 - Widget 590
 - width() 433
 - Wiederverwendung
 - durch Delegation 345
 - wildcard 657
 - Winterzeit 957
 - Wochentag 957
 - wofstream 919
 - Wrapperklasse für Iterator 899
 - write() 257, 428
 - ws 438
 - wstring 411, 919, 931
 - Wächter (Tabellenende) 93, 225
- X**
- XOR, bitweises 45
 - xvalue 480
- Y**
- year_month_day 385
 - yield() 552
- Z**
- z, Z (Suffix) 42
 - Zahl in String umwandeln 718
 - Zahlenbereich 41, 49
 - Zeichen 54
 - Zeichenkette 33, *siehe auch* String
 - C-String 227
 - Kopieren einer 231
 - Zeichenklasse (Regex) 538
 - Zeichensatz 54, 915
 - Zeiger 217, 234
 - Arithmetik 224
 - vs. Array 223
 - auf Basisklasse 318, 324
 - Darstellung
 - von [] 225
 - von [] [] 250
 - auf Elementdaten 266
 - auf Elementfunktionen 265
 - auf Funktionen 260
 - hängender 237
 - intelligente *siehe* Smart Pointer
 - Null-Zeiger 219
 - auf Oberklasse 310, 318
 - auf Objekt (Mehrfachvererbung) 335
 - auf lokale Objekte 221
 - Parameterübergabe per Z. 239
 - Zeile
 - einlesen *siehe* getline()
 - neue 55
 - Zeit-Server 619
 - Zeitkomplexität 986
 - Ziel (make) 652
 - Ziffernzeichen 54
 - Zufallszahlen 794
 - Zugriffsspezifizierer und -rechte 307
 - zusammengesetzte Datentypen 85
 - Zusicherung 143
 - Zustand 986
 - eines Iterators 464
 - Zuweisung 65, 69, 986
 - und Initialisierung 187
 - und Vererbung 310, 419
 - Zuweisungsoperator 188, 379, 496
 - implizite Deklaration 379, 419
 - und Vererbung 419
 - zweidimensionale Matrix 414
 - Zweierkomplement 41
 - Zwischenraumzeichen 105, 228, 425