

Errata

Bjarne Stroustrup: Eine Tour durch C++. Der praktische Leitfaden für modernes C++. 1. Auflage mitp 2023

In einigen Listings wurde fälschlicherweise anstelle eines Bindestrichs ein Fragezeichen angegeben. Nachfolgend stehen die korrigierten Listings.

In den E-Books wurden die Listings korrigiert.

Der Nachdruck wird die korrigierten Listings enthalten.

Kapitel 1

Seite 23, 3. Listing, 3. und 4. Zeile

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

Kapitel 2

Seite 43, 2. Listing, 5. Zeile

```
void f(Vector v, Vector& rv, Vector* pv)
{
    int i1 = v.sz;      // Zugriff über den Namen
    int i2 = rv.sz;    // Zugriff über eine Referenz
    int i3 = pv->sz;    // Zugriff über einen Zeiger
}
```

Seite 49, 1. Listing, 13. und 14. Zeile

```
enum class Type { ptr, num }; // ein Type kann die Werte ptr und
num enthalten (§2.4)

struct Entry {
    string name; // string ist ein Standardbibliothekstyp
    Type t;
```

```

    Node* p;           // benutzt p, falls t==Type::ptr
    int i;            // benutzt i, falls t==Type::num
};

void f(Entry* pe)
{
    if (pe->t == Type::num)
        cout << pe->i;
    // ...
}

```

Seite 49f., 3. Listing, 10. und 11. Zeile

```

struct Entry {
    string name;
    Type t;
    Value v; // v.p soll benutzt werden, falls t==Type::ptr;
             // v.i soll benutzt werden, falls t==Type::num
};

void f(Entry* pe)
{
    if (pe->t == Type::num)
        cout << pe->v.i;
    // ...
}

```

Seite 50, 2. Listing, 8. und 10. Zeile

```

struct Entry {
    string name;
    variant<Node*,int> v;
};

void f(Entry* pe)
{
    if (holds_alternative<int>(pe->v)) // enthält *pe einen int?
                                        // (siehe §15.4.1)
        cout << get<int>(pe->v);      // holt das int
}

```

```
// ...  
}
```

Kapitel 3

Seite 69, 2. Listing, 1. Zeile

```
auto mul(int i, double d) -> double { return i*d; } // der Rückgabety  
// ist "double"
```

Seite 69, 2. Listing, 1., 2. und 3. Zeile

```
auto next_elem() -> Elem*;  
auto exit(int) -> void;  
auto sqrt(double) -> double;
```

Kapitel 4

Seite 76, 1. Listing, 1. Zeile

```
Vector v(-27);
```

Seite 76f., 3. Listing, 16. Zeile

```
void test(int n)  
{  
    try {  
        Vector v(n);  
    }  
    catch (std::length_error& err) {  
        // ... befasst sich mit negativer Größe ...  
    }  
    catch (std::bad_alloc& err) {  
        // ... befasst sich mit fehlendem Speicher ...  
    }  
}  
  
void run()  
{  
    test(-27); // wirft length_error (-27 ist zu klein)}
```

```
test(1'000'000'000); // könnte bad_alloc werfen
test(10);           // wahrscheinlich okay
}
```

Kapitel 5

Seite 89f., 3. Listing, 23., 25. und 26. Zeile

```
class complex {
    double re, im;           // Repräsentation: zwei Doubles
public:
    complex(double r, double i) :re{r}, im{i} {} // konstruiert
                                   // complex aus zwei Skalaren
    complex(double r) :re{r}, im{0} {}           // konstruiert
                                   // complex aus einem Skalar
    complex() :re{0}, im{0} {} // Standard complex: {0,0}

    double real() const { return re; }
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }

    complex& operator+=(complex z)
    {
        re+=z.re;           // addiert zu re und im
        im+=z.im;
        return *this;      // gibt das Ergebnis zurück
    }

    complex& operator-=(complex z)
    {
        re-=z.re;
        im-=z.im;
        return *this;
    }

    complex& operator*=(complex);           // irgendwo außerhalb der
                                           // Klasse definiert
}
```

```

    complex& operator/=(complex);           // irgendwo außerhalb der
                                           // Klasse definiert
};

```

Seite 91, 2. Listing, 2. und 3. Zeile

```

complex operator+(complex a, complex b) { return a+=b; }
complex operator-(complex a, complex b) { return a-=b; }
complex operator-(complex a) { return {-a.real(), -a.imag()}; }
                                           // unäres Minus; die Negierung
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }

```

Seite 91f., 4. Listing, 7. Zeile

```

void f(complex z)
{
    complex a {2.3};           // konstruiert {2.3,0.0} aus 2.3
    complex b {1/a};
    complex c {a+z*complex{1,2.3}};
    if (c != b)
        c = -(b/a)+2*b;
}

```

Seite 99, 2. Listing, 20. Zeile

```

class List_container : public Container {           // List_container
                                                    // implementiert Container
public:
    List_container() { }           // Leere Liste
    List_container(initializer_list<double> il) : ld{il} { }
    ~List_container() {}

    double& operator[](int i) override;
    int size() const override { return ld.size(); }
private:
    std::list<double> ld;           // (Standardbibliothek) Liste aus doubles
                                    // (§12.3)
};

```

```

double& List_container::operator[](int i)
{
    for (auto& x : ld) {
        if (i==0)
            return x;
        --i;
    }
    throw out_of_range{"List container"};
}

```

Seite 102, 2. Listing, 5. Zeile

```

void rotate_all(vector<Shape*>& v, int angle)    // dreht die Elemente
                                                // von v um angle Grad
{
    for (auto p : v)
        p->rotate(angle);
}

```

Seite 104, 2. Listing, 5. und 6. Zeile

```

void Smiley::draw() const
{
    Circle::draw();
    for (auto p : eyes)
        p->draw();
    mouth->draw();
}

```

Seite 105f. , 1. Listing, 19., 20. und 21. Zeile

```

enum class Kind { circle, triangle, smiley };

Shape* read_shape(istream& is)    // liest Beschreibung der Formen aus
                                  // dem Eingabestrom is
{
    // ... liest Formen-Header aus is und bestimmt die Art der Form, k ...

    switch (k) {
    case Kind::circle:
        // ... liest Kreisdaten {Point,int} in p und r ein ...
        return new Circle{p,r};
    }
}

```

```

        case Kind::triangle:
// ... liest Dreiecksdaten {Point,Point,Point} in p1, p2 und p3 ein ...
            return new Triangle{p1,p2,p3};
        case Kind::smiley:
// ... liest Smiley-Daten {Point,int,Shape,Shape,Shape} in p, r, e1, e2
// und m ein ...
            Smiley* ps = new Smiley{p,r};
            ps->add_eye(e1);
            ps->add_eye(e2);
            ps->set_mouth(m);
            return ps;
    }
}

```

Kapitel 6

Seite 128, 2. Listing, 4. Zeile

```

class Vector {
public:
    Vector(int s);    // Konstruktor: etabliert Invariante,
                    //beschafft Ressourcen
    ~Vector() { delete[] elem; } // Destruktor: gibt Ressourcen frei

    Vector(const Vector& a);           // Kopier-Konstruktor
    Vector& operator=(const Vector& a); // Kopier-Zuweisung
}

```

Seite 128, 2. Listing, 4. Zeile

```

struct R2 {
    int m;
    auto operator<=>(const R2& a) const {
        return a.m == m ? 0 : a.m < m ? -1 : 1;
    }
};

```

Kapitel 7

Seite 135, 1. Listing, 9. Zeile

```
template<typename T>
class Vector {
private:
    T* elem;    // elem zeigt auf ein Array aus sz Elementen des Typs T
    int sz;
public:
    explicit Vector(int s);    // Konstruktor: etabliert Invariante,
                               //beschafft Ressourcen

    ~Vector() { delete[] elem; } // Destruktor: gibt Ressourcen frei

    // ... Kopier- und Verschiebe-Operationen ...
};
```

Seite 142, 2. Listing, 2. Zeile

```
template<typename Iter>
    Vector(Iter, Iter) -> Vector<typename Iter::value_type>;
```

Seite 148, 1. Listing, 4. Zeile

```
template<class S>
void rotate_and_draw(vector<S>& v, int r)
{
    for_each(v, [] (auto& s) { s->rotate(r); s->draw(); });
}
```

Seite 148, 2. Listing, 1. Zeile

```
for_each(v, [] (Pointer_to_class auto& s) { s->rotate(r); s->draw(); });
```

Seite 152, 1. Listing, 5. Zeile

```
template <class T>
    constexpr T viscosity = 0.4;

template <class T>
    constexpr space_vector<T> external_acceleration = { T{}, T{-9.8}, T{} };

auto vis2 = 2*viscosity<double>;
auto acc = external_acceleration<float>;
```

Seite 152, 2. Listing, 4. Zeile

```
template <class F>
struct Final_action {
    explicit Final_action(F f) :act(f) {}
    ~Final_action() { act(); }
    F act;
};
```

Kapitel 8

Seite 160f., 4. Listing, 4. Zeile

```
template<forward_iterator Iter>
void advance(Iter p, int n) // verschiebt p um n Elemente vorwärts
{
    while (n-->0)
        ++p; // ein Vorwärtsiterator hat ++, aber nicht + oder +=
}

template<random_access_iterator Iter>
void advance(Iter p, int n) // verschiebt p um n Elemente vorwärts
{
    p+=n; // ein Random-Access-Iterator hat +=
}
```

Seite 163, 1. Listing, 4. und 5. Zeile

```
template<typename T>
concept Equality_comparable =
    requires (T a, T b) {
        { a == b } -> Boolean; // vergleicht Ts mit ==
        { a != b } -> Boolean; // vergleicht Ts mit !=
    };
```

Seite 164, 1. Listing, 4., 6., 8. und 10. Zeile

```
template<typename T, typename T2 = T>
concept Equality_comparable =
    requires (T a, T2 b) {
        { a == b } -> Boolean; // vergleicht ein T mit einem T2
    };
```

```

// mittels ==
{ a != b } -> Boolean; // vergleicht ein T mit einem T2
// mittels !=
{ b == a } -> Boolean; // vergleicht ein T2 mit einem T
// mittels ==
{ b != a } -> Boolean; // vergleicht ein T2 mit einem T
// mittels !=
};

```

Seite 164, 3. Listing, 5. und 6. Zeile

```

template<typename T, typename U = T>
concept Number =
    requires(T x, U y) { // Etwas mit arithmetischen Operationen und
                        // einer Null
        x+y; x-y; x*y; x/y;
        x+=y; x-=y; x*=y; x/=y;
        x=x; // kopieren
        x=0;
    }

```

Seite 165, 2. Listing, 6. und 8. Zeile

```

template<typename S>
concept Sequence = requires (S a) {
    typename range_value_t<S>; // S muss einen Werttyp haben
    typename iterator_t<S>; // S muss einen Iterortyp haben

    { a.begin() } -> same_as<iterator_t<S>>; // S muss ein begin()
                                           // haben, das einen Iterator zurückgibt
    { a.end() } -> same_as<iterator_t<S>>;

    requires input_iterator<iterator_t<S>>; // der Iterator von S<
                                           // muss ein input_iterator sein
    requires same_as<range_value_t<S>, iter_value_t<S>>;
};

```

Kapitel 9

Seite 182, 2. Listing, 5. Zeile

```
#include<string> // macht die Standard-String-Infrastruktur zugänglich
using namespace std; // macht die std-Namen ohne das Präfix std::
                    // zugänglich

string s {"C++ is a general-purpose programming language"}; // Okay:
                                                            // string ist std::string
```

Kapitel 10

Seite 187, 1. Listing, 6. Zeile

```
string compose(const string& name, const string& domain)
{
    return name + '@' + domain;
}

auto addr = compose("dmr", "bell-labs.com");
```

Seite 193, 2. Listing, 1. Zeile

```
regex pat {R"(\w{2}\s*\d{5}(\-\d{4})?)"}; // Muster der US-Postleitzahlen:
                                         // XXdddd-dddd und Varianten
```

Seite 193, 4. Absatz, 2. Zeile

Für Leute, die bereits reguläre Ausdrücke in irgendeiner Sprache benutzt haben, wird `\w{2}\s*\d{5}(\-\d{4})?` vertraut aussehen.

Seite 193, 3. Listing, 1. Zeile

```
regex pat {"\\w{2}\\s*\\d{5}(-\\d{4})?"}; // Muster der US-Postleitzahlen
```

Seite 194f., 2. Listing, 9. Zeile

```
void use()
{
    ifstream in("file.txt"); // Eingabedatei
    if (!in) { // prüft, ob die Datei geöffnet wurde
        cerr << "no file\n";
        return;
    }
}
```

```

}

regex pat {R"(\w{2}\s*\d{5})(-\d{4})?"); // Muster der
                                         // US-Postleitzahlen

int lineno = 0;
for (string line; getline(in,line); ) {
    ++lineno;
    smatch matches; // passende Strings kommen hierher
    if (regex_search(line, matches, pat)) {
        cout << lineno << ": " << matches[0] << '\n'; // der
                                                         // komplette Treffer
        if (1<matches.size() && matches[1].matched) // falls
                                                    // es ein Untermuster gibt und falls dies passt
            cout << "\t: " << matches[1] << '\n'; // Untertreffer
    }
}
}

```

Seite 196, 4. Listing, 1., 2. und 3. Zeile

```

\d+ \d+ // keine Untermuster
\d+ (-\d+) // ein Untermuster
(\d+) (-\d+) // zwei Untermuster

```

Seite 199, 3. Listing, 2. und 3. Zeile

```

Ax* // A, Ax, Axxxx
Ax+ // Ax, AxxxNicht A
\d?\d // 1-2, 12 Nicht 1--2
\w{2}-\d{4,5} // Ab-1234, XX-54321, 22-5432 Ziffern sind in \w
(\d*:)?(\d+) // 12:3, 1:23, 123, :123 Nicht 123:
(bs|BS) // bs, BS Nicht bS
[aeiouy] // a, o, u Ein englischer Vokal, nicht x
[^aeiouy] // x, k Kein englischer Vokal, nicht e
[a^eiouy] // a, ^, o, u Ein englischer Vokal oder ^

```

Kapitel 11

Seite 212, 5. Listing, 1. und 2. Zeile

```
birthday: 2021-11-28  
zt: 2021-12-05 11:03:13.5945638 EST
```

Kapitel 12

Seite 231f., 3. Listing, 10. und 11. Zeile

```
template<typename T>  
class Vector {  
    allocator<T> alloc;    // Allokator aus der Standardbibliothek für  
                        // Platz für die Ts  
    T* elem;    // Zeiger auf das erste Element  
    T* space;    // Zeiger auf den ersten (und uninitialisierten) Slot  
    T* last;    // Zeiger auf den letzten Slot  
public:  
    // ...  
    int size() const { return space - elem; }    // Anzahl der Elemente  
    int capacity() const { return last - elem; }    // Anzahl der  
                        // verfügbaren Slots für die Elemente  
    // ...  
    void reserve(int newsz);    // vergrößert capacity() auf newsz  
    // ...  
    void push_back(const T& t);    // kopiert t in Vector  
    void push_back(T&& t);    // verschiebt t in Vector  
};
```

Seite 237, 2. Listing, 4. und 5. Zeile

```
int get_number(const string& s)  
{  
    for (auto p = phone_book.begin(); p != phone_book.end(); ++p)  
        if (p->name == s)  
            return p->number;  
    return 0;    // verwendet 0 für "Nummer nicht gefunden"  
}
```

Kapitel 13

Seite 250f., 1. Listing, 20. und 22. Zeile

```
template<typename C>
class Checked_iter {
public:
    using value_type = typename C::value_type;
    using difference_type = int;

    Checked_iter() { throw Missing_container{}; }
    // Das forward_iterator-Konzept erfordert einen Default-Konstruktor
    Checked_iter(C& cc) : pc{ &cc } {}
    Checked_iter(C& cc, typename C::iterator pp) : pc{ &cc }, p{ pp } {}

    Checked_iter& operator++() { check_end(); ++p; return *this; }
    Checked_iter operator++(int) { check_end(); auto t{ *this };
    ++p; return t; }
    value_type& operator*() const { check_end(); return *p; }

    bool operator==(const Checked_iter& a) const { return p==a.p; }
    bool operator!=(const Checked_iter& a) const { return p!=a.p; }
private:
    void check_end() const { if (p == pc->end()) throw Overflow{}; }
    C* pc {}; // initialisiert standardmäßig auf nullptr
    typename C::iterator p = pc->begin();
};
```

Kapitel 14

Seite 272, 2. Tabelle, 9. Zeile

Kernsprachkonzepte <concepts>

T und U sind Typen.

same_as<T,U>	T ist dasselbe wie U.
derived_from<T,U>	T ist abgeleitet von U.
convertible_to<T,U>	Ein T kann in ein U umgewandelt werden.
common_reference_with<T,U>	T und U haben einen gemeinsamen Referenztyp.

<code>common_with<T,U></code>	T und U haben einen gemeinsamen Typ.
<code>integral<T></code>	T ist ein integraler Typ.
<code>signed_integral<T></code>	T ist ein vorzeichenbehafteter integraler Typ.
<code>unsigned_integral<T></code>	T ist ein vorzeichenloser integraler Typ.
<code>floating_point<T></code>	T ist ein Gleitkommatyp.
<code>assignable_from<T,U></code>	Ein U kann einem T zugewiesen werden.
<code>swappable_with<T,U></code>	Ein T kann mit einem U vertauscht werden.
<code>swappable<T></code>	<code>swappable_with<T,T></code>

Kapitel 15

Seite 299, 2. Listing, 7. Zeile

```
template<class... Ts>
struct overloaded : Ts... { // variadisches Template (§8.4)
    using Ts::operator()...;
};

template<class... Ts>
    overloaded(Ts...) -> overloaded<Ts...>; // Deduktionsanleitung
```

Kapitel 16

Seite 305f., 1. Listing, 8., 9. und 11. Zeile

```
using namespace std::chrono; // im Unterraum std::chrono; siehe §3.3

auto t0 = system_clock::now();
do_work();
auto t1 = system_clock::now();

cout << t1-t0 << "\n"; // standardmäßige Einheit: 20223[1/1000000]s
cout << duration_cast<milliseconds>(t1-t0).count() << "ms\n";
    // festgelegte Einheit: 2ms
cout << duration_cast<nanoseconds>(t1-t0).count() << "ns\n";
    // festgelegte Einheit: 2022300ns
```

Seite 309, 1. Listing, 3. Zeile

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(),v.end(),[](Shape* p) { p->draw(); });
}
```

Seite 310, 1. Listing, 7. Zeile

```
int f1(double);
function<int(double)> fct1 {f1}; // initialisiert auf f1

int f2(string);
function fct2 {f2};           // Typ von fct2 ist function<int(string)>

function fct3 = [](Shape* p) { p->draw(); }; // Typ von fct3 ist
                                           // function<void(Shape*)>
```

Seite 314, 3. Listing, 5. Zeile

```
template<class T>
void cpy1(T* first, T* last, T* target)
{
    if constexpr (is_trivially_copyable_v<T>)
        memcpy(first, target, (last - first) * sizeof(T));
    else
        while (first != last) *target++ = *first++;
}
```

Seite 314f., 4. Listing, 5. Zeile

```
template<typename T>
class Smart_pointer {
    // ...
    T& operator*() const;
    T* operator->() const; // -> sollte funktionieren, falls und
                          // tatsächlich nur falls T eine Klasse ist
};
```

Seite 315, 2. Listing, 5. Zeile

```
template<typename T>
class Smart_pointer {
```

```

// ...
T& operator*() const;
T* operator->() const requires is_class v<T>; // -> wird definiert,
// falls und tatsächlich nur falls T eine Klasse ist
};

```

Seite 315, 3. Listing, 8. Zeile

```

template<typename T>
concept Class = is_class v<T> || is_union_v<T>; // Unions sind Klassen

template<typename T>
class Smart_pointer {
    // ...
    T& operator*() const;
    T* operator->() const requires Class<T>; // -> wird definiert,
    // falls und tatsächlich nur falls T eine Klasse oder eine Union ist
};

```

Seite 316, 1. Listing, 5. Zeile

```

template<typename T>
class Smart_pointer {
    // ...
    T& operator*();
    enable_if<is_class_v<T>,T&> operator->(); // -> wird definiert,
    // falls und tatsächlich nur falls T eine Klasse ist
};

```

Kapitel 17

Seite 326, 1. Listing, 2. Zeile

```

errno = 0; // löscht den alten Fehlerstatus
double d = sqrt(-1);
if (errno==EDOM)
    cerr << "sqrt() not defined for negative argument\n";

errno = 0; // löscht den alten Fehlerstatus
double dd = pow(numeric_limits<double>::max(),2);
if (errno == ERANGE)

```

```
cerr << "result of pow() too large to represent as a double\n";
```

Kapitel 18

Seite 345, 2. Listing, 8. Zeile

```
using namespace chrono;           // siehe §16.2.1

auto t0 = high_resolution_clock::now();
this_thread::sleep_for(milliseconds{20});
auto t1 = high_resolution_clock::now();

cout << duration_
cast<nanoseconds>(t1-t0).count() << " nanoseconds passed\n";
```

Seite 353, 1. Listing, 1. und 12. Zeile

```
atomic<int> result = 1;           // setzt einen resultierenden Index hierhin

template<class T> struct Range { T* first; T* last; };           // eine
                                                                    // Möglichkeit, einen Bereich aus Ts zu übergeben

void find(stop_token tok, const string* base,
const Range<string> r, const string target)
{
    for (string* p = r.first; p!=r.last && !tok.stop_requested(); ++p)
        if (match(*p, target)) {           // match() wendet irgendwelche
                                                    // Vergleichskriterien auf die zwei Strings an
                result = p - base; // der Index des gefundenen Elements
            return;
        }
}
```

Seite 353, 2. Listing, 14. Zeile

```
void find_all(vector<string>& vs, const string& key)
{
    int mid = vs.size()/2;
    string* pvs = &vs[0];

    stop_source ssl{};
```

```
jthread t1(find, ss1.get_token(), pvs, Range{pvs,pvs+mid}, key);
           // erste Hälfte der vs

stop_source ss2{};
jthread t2(find, ss2.get_token(), pvs, Range{pvs+mid,pvs+vs.size()} , key);
           // zweite Hälfte der vs

while (result == 1)
    this_thread::sleep_for(10ms);

ss1.request_stop(); // wir haben ein Ergebnis: alle Threads stoppen!
ss2.request_stop();

// ... Ergebnis benutzen ...
}
```