

Eine Tour durch C++

Die kurze Einführung in den neuen Standard C++11

Bearbeitet von
Bjarne Stroustrup, Frank Langenau

1. Auflage 2015. Buch. ca. 200 S.
ISBN 978 3 446 43962 7
Format (B x L): 20,2 x 24,6 cm
Gewicht: 609 g

[Weitere Fachgebiete > EDV, Informatik > Programmiersprachen: Methoden > Programmier- und Skriptsprachen](#)

schnell und portofrei erhältlich bei


DIE FACHBUCHHANDLUNG

Die Online-Fachbuchhandlung beck-shop.de ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.



Leseprobe

Bjarne Stroustrup

Eine Tour durch C++

Die kurze Einführung in den neuen Standard C++11

Übersetzt aus dem Englischen von Frank Langenau

ISBN (Buch): 978-3-446-43962-7

Weitere Informationen oder Bestellungen unter

<http://www.hanser-fachbuch.de/978-3-446-43962-7>

sowie im Buchhandel.

Inhalt

Vorwort	XI
1 Die Grundlagen	1
1.1 Einführung	1
1.2 Programme	1
1.3 Hello, World!	2
1.4 Funktionen	3
1.5 Typen, Variablen und Arithmetik	5
1.6 Gültigkeitsbereich und Lebensdauer	8
1.7 Konstanten	9
1.8 Zeiger, Arrays und Referenzen	10
1.9 Tests	13
1.10 Ratschläge	15
2 Benutzerdefinierte Typen	17
2.1 Einführung	17
2.2 Strukturen	17
2.3 Klassen	19
2.4 Unions	21
2.5 Aufzählungen	22
2.6 Ratschläge	23
3 Modularität	25
3.1 Einführung	25
3.2 Separate Kompilierung	26
3.3 Namespaces	28
3.4 Fehlerbehandlung	29
3.4.1 Ausnahmen	29
3.4.2 Invarianten	30
3.4.3 Statische Assertionen	32
3.5 Ratschläge	33

4	Klassen	35
4.1	Einführung	35
4.2	Konkrete Typen	36
4.2.1	Ein arithmetischer Typ	37
4.2.2	Ein Container	38
4.2.3	Container initialisieren	40
4.3	Abstrakte Typen	41
4.4	Virtuelle Funktionen	44
4.5	Klassenhierarchien	45
4.5.1	Explizites Überschreiben	47
4.5.2	Vorzüge von Hierarchien	48
4.5.3	Navigation in Hierarchien	50
4.5.4	Ressourcenlecks vermeiden	50
4.6	Kopieren und Verschieben	51
4.6.1	Container kopieren	52
4.6.2	Container verschieben	53
4.6.3	Wichtige Operationen	55
4.6.4	Ressourcenverwaltung	57
4.6.5	Operationen unterdrücken	59
4.7	Ratschläge	59
5	Templates	63
5.1	Einführung	63
5.2	Parametrisierte Typen	63
5.3	Funktions-Templates	65
5.4	Konzepte und generische Programmierung	66
5.5	Funktionsobjekte	68
5.6	Variadische Templates	70
5.7	Alias	71
5.8	Modell der Template-Kompilierung	72
5.9	Ratschläge	73
6	Überblick über die Bibliothek	75
6.1	Einführung	75
6.2	Komponenten der Standardbibliothek	76
6.3	Header und Namespace der Standardbibliothek	77
6.4	Ratschläge	78
7	Strings und reguläre Ausdrücke	79
7.1	Einführung	79
7.2	Strings	79
7.2.1	Eine string -Implementierung	81

7.3	Reguläre Ausdrücke	82
7.3.1	Suchen	83
7.3.2	Notation regulärer Ausdrücke	84
7.3.3	Iteratoren	88
7.4	Ratschläge	89
8	E/A-Streams	91
8.1	Einführung	91
8.2	Ausgabe	92
8.3	Eingabe	93
8.4	E/A-Status	95
8.5	Ein-/Ausgabe von benutzerdefinierten Typen	96
8.6	Formatierung	97
8.7	File Streams	98
8.8	String-Streams	99
8.9	Ratschläge	100
9	Container	103
9.1	Einführung	103
9.2	vector	103
9.2.1	Elemente	106
9.2.2	Bereichsüberprüfung	106
9.3	list	108
9.4	map	109
9.5	unordered_map	110
9.6	Überblick über Container	111
9.7	Ratschläge	113
10	Algorithmen	115
10.1	Einführung	115
10.2	Iteratoren verwenden	116
10.3	Iteratortypen	119
10.4	Stream-Iteratoren	120
10.5	Prädikate	122
10.6	Überblick über Algorithmen	122
10.7	Containeralgorithmen	123
10.8	Ratschläge	124
11	Utilities	125
11.1	Einführung	125
11.2	Ressourcenverwaltung	125
11.2.1	unique_ptr und shared_ptr	126

11.3	Spezialisierte Container	129
11.3.1	array	130
11.3.2	bitset	131
11.3.3	pair und tuple	132
11.4	Zeit	134
11.5	Funktionsadapter	134
11.5.1	bind()	135
11.5.2	mem_fn()	135
11.5.3	function	136
11.6	Typfunktionen	137
11.6.1	iterator_traits	138
11.6.2	Typprädikate	140
11.7	Ratschläge	140
12	Numerik	143
12.1	Einführung	143
12.2	Mathematische Funktionen	143
12.3	Numerische Algorithmen	144
12.4	Komplexe Zahlen	145
12.5	Zufallszahlen	146
12.6	Vektorarithmetik	148
12.7	Numerische Grenzen	148
12.8	Ratschläge	149
13	Nebenläufigkeit	151
13.1	Einführung	151
13.2	Tasks und Threads	152
13.3	Argumente übergeben	153
13.4	Ergebnisse zurückgeben	154
13.5	Daten gemeinsam nutzen	154
13.6	Warten auf Ereignisse	156
13.7	Kommunizierende Tasks	158
13.7.1	future und promise	158
13.7.2	packaged_task	159
13.7.3	async()	160
13.8	Ratschläge	161
14	Geschichte und Kompatibilität	163
14.1	Historische Anmerkungen	163
14.1.1	Chronik	164
14.1.2	Die frühen Jahre	165
14.1.3	Die ISO-C++-Standards	167

14.2	C++11-Erweiterungen	169
14.2.1	Sprachfeatures	169
14.2.2	Komponenten der Standardbibliothek	170
14.2.3	Veraltete Features	171
14.2.4	Typumwandlungen	172
14.3	C/C++-Kompatibilität	173
14.3.1	C und C++ sind Geschwister	173
14.3.2	Kompatibilitätsprobleme	175
14.3.2.1	Stilprobleme	175
14.3.2.2	void*	177
14.3.2.3	C++-Schlüsselwörter	177
14.3.2.4	Bindung	178
14.4	Literaturhinweise	178
14.5	Ratschläge	181
Index	183

Vorwort

*When you wish to instruct,
be brief.
– Cicero*

C++ fühlt sich wie eine neue Sprache an. Das heißt, ich kann meine Ideen in C++11 klarer, einfacher und direkter ausdrücken als ich es in C++98 konnte. Darüber hinaus werden die resultierenden Programme durch den Compiler besser überprüft und laufen schneller.

Wie andere moderne Sprachen ist C++ groß und es gibt eine große Anzahl von Bibliotheken, die für den effizienten Einsatz dieser Sprache erforderlich sind. Dieses kompakte Buch soll einem erfahrenen Programmierer eine Vorstellung davon vermitteln, was modernes C++ ausmacht. Das Buch behandelt die wichtigsten Sprachfeatures und die Hauptkomponenten der Standardbibliothek. Es lässt sich zwar in wenigen Stunden lesen, doch gehört offenbar wesentlich mehr dazu, gutes C++ zu schreiben, als man an einem Tag lernen kann. Allerdings ist hier nicht Meisterschaft das Ziel. Vielmehr soll das Buch einen Überblick bieten, Prinzipbeispiele angeben und einem Programmierer bei den ersten Schritten helfen. Um das Thema zu beherrschen, sollten Sie sich mein Buch *The C++ Programming Language, Fourth Edition* (TC++PL4) [Stroustrup, 2013] ansehen. Letztlich ist dieses Buch eine erweiterte Version des Stoffes, der die Kapitel 2 bis 5 von TC++PL4 ausmacht, auch *Tour of C++* genannt. Um dieses Buch zu einem einigermaßen eigenständigen Werk zu machen, habe ich Erweiterungen und Verbesserungen hinzugefügt. Die Struktur dieser Tour folgt der von TC++PL4, sodass Sie ergänzenden Stoff leicht finden können. Auch die Übungen für TC++PL4 auf meiner Website (www.stroustrup.com) sind geeignet, um diese Tour zu unterstützen.

Es wird hier davon ausgegangen, dass Sie bereits programmiert haben. Andernfalls sollten Sie sich zuerst mit einem Lehrbuch wie zum Beispiel *Programming: Principles and Practice Using C++* [Stroustrup, 2009] befassen, bevor Sie hier fortfahren. Doch auch wenn Sie bereits programmiert haben, können die von Ihnen verwendete Sprache oder die Anwendungen, die Sie geschrieben haben, erheblich von dem hier vorgestellten C++-Stil abweichen.

Stellen Sie sich als Analogie eine kurze Stadtrundfahrt vor, etwa in Kopenhagen oder New York. In nur wenigen Stunden erhaschen Sie einen kurzen Blick auf die Hauptattraktionen, erfahren einige Hintergrundgeschichten und bekommen normalerweise einige Vorschläge, was Sie sich als Nächstes ansehen sollten. Nach einer solchen Tour kennen Sie die Stadt aber *nicht*. Und Sie verstehen auch *nicht*, was Sie alles gesehen und gehört haben. Auch werden Sie sich *nicht* in den formellen und informellen Regeln zurechtfinden, die das Leben in der Stadt bestimmen. Um eine Stadt wirklich kennenzulernen, müssen Sie in ihr leben – am besten für mehrere Jahre. Mit ein bisschen Glück allerdings haben Sie einen kleinen

Überblick gewonnen, eine Vorstellung davon, was die Stadt Besonderes zu bieten hat, und Anregungen, was für Sie interessant sein könnte. Nach der Tour kann die eigentliche Erkundung beginnen.

Diese Tour stellt die wesentlichen Sprachfeatures von C++ in dem Maße vor, wie sie Programmierstile unterstützen, beispielsweise die objektorientierte und generische Programmierung. Sie versucht nicht, eine detaillierte Sicht der Sprache Feature für Feature in der Art eines Referenzhandbuchs zu bieten. Analog dazu stellt sie die Standardbibliotheken in Form von Beispielen vor, ohne vollständig sein zu wollen. Außerdem werden keine Bibliotheken beschrieben, die über die im ISO-Standard definierten hinausgehen. Der Leser kann bei Bedarf nach unterstützendem Material recherchieren. Beispiele hierfür sind [Stroustrup, 2009] und [Stroustrup, 2013], doch im Web stehen enorme Mengen an weiterführenden Informationen (in unterschiedlicher Qualität) zur Verfügung. Wenn ich zum Beispiel eine Funktion oder Klasse der Standardbibliothek erwähne, lässt sich deren Definition leicht auffinden. Und wenn Sie sich die Dokumentationen der jeweiligen Header ansehen (die ebenfalls im Web zugänglich sind), werden Sie auch auf zahlreiche verwandte Elemente stoßen.

Diese Tour hier stellt C++ als geschlossene Einheit vor und nicht in Themen aufgegliedert. Deshalb kennzeichnet sie die Sprachfeatures auch nicht danach, ob sie in C vorhanden, Teil von C++98 oder neu in C++11 sind. Solche historischen Informationen finden Sie in Kapitel 14.

Danksagungen

Ein großer Teil des hier präsentierten Stoffes stammt aus „The C++ Programming Language“ (TC++PL4) [Stroustrup, 2013], 2015 auf Deutsch erschienen unter dem Titel „Die C++-Programmiersprache“. Ich danke deshalb allen, die zu diesem Buch beigetragen haben. Außerdem geht mein Dank an Peter Gordon, der zuerst vorgeschlagen hatte, die vier Tour-Kapitel aus TC++PL4 zu einer einigermaßen eigenständigen und konsistenten eigenen Publikation zu erweitern.

College Station, Texas

Bjarne Stroustrup

5

Templates

Your quote here.

– B. Stroustrup

■ 5.1 Einführung

Wer einen Vektor verwenden möchte, wird sicherlich nicht immer einen Vektor mit **double**-Typen brauchen. Ein Vektor ist ein allgemeines Konzept, das unabhängig vom Begriff einer Gleitkommazahl ist. Folglich sollte der Elementtyp eines Vektors unabhängig dargestellt werden. Ein *Template* (Vorlage) ist eine Klasse oder eine Funktion, die wir mit einem Satz von Typen oder Werten parametrisieren. Mit Templates stellen wir Konzepte dar, die man am besten als etwas sehr Allgemeines betrachtet, aus dem sich spezifische Typen und Funktionen erzeugen lassen, indem man Argumente – wie zum Beispiel den Elementtyp **double** – angibt.

■ 5.2 Parametrisierte Typen

Unseren Vektor-Typ für **double**-Werte können wir zu einem Vektor für einen beliebigen Typ verallgemeinern, indem wir ihn zu einem Template machen und den konkreten Typ **double** durch einen Parameter ersetzen. Zum Beispiel:

```
template<typename T>
class Vector {
private:
    T* elem;    // elem zeigt auf ein Array von sz Elementen des Typs T
    int sz;
public:
    explicit Vector(int s);        // Konstruktor: Invariante einrichten,
                                   // Ressourcen belegen
    ~Vector() { delete[] elem; }  // Destruktor: Ressourcen freigeben

    // ... Kopier- und Verschiebeoperationen ...

    T& operator[](int i);
    const T& operator[](int i) const;
    int size() const { return sz; }
};
```

Das Präfix `template<typename T>` macht `T` zu einem Parameter der Deklaration, vor der er steht. Das ist die Version von C++ für den mathematischen Ausdruck „für alle `T`“ oder genauer „für alle Typen `T`“. Ein Typparameter lässt sich mit `class` genauso wie mit `typename` einführen und in älterem Code ist oftmals `template<class T>` als Präfix zu sehen.

Die Member-Funktionen können ähnlich definiert sein:

```
template<typename T>
Vector<T>::Vector(int s)
{
    if (s<0)
        throw Negative_size{};
    elem = new T[s];
    sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

Mit diesen Definitionen können wir wie folgt **Vector**-Objekte definieren:

```
Vector<char> vc(200);           // Vektor mit 200 Zeichen
Vector<string> vs(17);         // Vektor mit 17 Strings
Vector<list<int>> vli(45);      // Vektor mit 45 Listen von Ganzzahlen
```

Der String `>>` in **Vector<list<int>>** schließt die verschachtelten Template-Argumente ab; es handelt sich also nicht um einen falsch gesetzten Eingabeoperator. Zwischen den beiden spitzen Klammern ist kein Leerzeichen erforderlich (wie noch in C++98).

Die **Vector**-Objekte können wir so verwenden:

```
void write(const Vector<string>& vs)    // Vektor von Strings
{
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

Um die bereichsbasierte **for**-Schleife für unseren **Vector** zu unterstützen, müssen wir geeignete **begin()**-und **end()**-Funktionen definieren:

```
template<typename T>
T* begin(Vector<T>& x)
{
    return x.size() ? &x[0] : nullptr;    // Zeiger auf erstes Element oder nullptr
}

template<typename T>
T* end(Vector<T>& x)
{
    return begin(x)+x.size();             // Zeiger hinter das letzte Element
}
```

Damit können wir schreiben:

```
void f2(Vector<string>& vs)    // Vektor von Strings
{
    for (auto& s : vs)
        cout << s << '\n';
}
```

In ähnlicher Weise können wir Listen, Vektoren, Maps (d.h. assoziative Arrays) usw. als Templates definieren (Kapitel 9).

Da Templates zur Übersetzungszeit verarbeitet werden, bringen sie verglichen mit „hand-gestricktem Code“ keinen Overhead zur Laufzeit mit sich. Tatsächlich ist der Code, der für **Vector<double>** generiert wird, identisch mit dem Code, der für die Version von **Vector** aus Kapitel 4 generiert wird. Darüber hinaus ist der Code, der für **vector<double>** der Standardbibliothek generiert wird, wahrscheinlich besser (da mehr Aufwand in seine Implementierung geflossen ist).

Außer Typargumenten kann ein Template auch Wertargumente übernehmen. Zum Beispiel:

```
template<typename T, int N>
struct Buffer {
    using value_type = T;
    constexpr int size() { return N; }
    T[N];
    // ...
};
```

Der Alias (**value_type**) und die **constexpr**-Funktion sind dafür vorgesehen, dass Benutzer auf die Template-Argumente (nur lesend) zugreifen können.

Wertargumente sind in vielen Kontexten nützlich. Zum Beispiel ist es mit **Buffer** möglich, Puffer beliebiger Größe ohne den bei Verwendung von Freispeicher (dynamischem Speicher) entstehenden Overhead zu erzeugen:

```
Buffer<char,1024> glob;    // globaler Zeichenpuffer (statisch alloziert)

void fct()
{
    Buffer<int,10> buf;    // lokaler Ganzzahlpuffer (auf dem Stack)
    // ...
}
```

Ein Template-Wertargument muss ein konstanter Ausdruck sein.

■ 5.3 Funktions-Templates

Templates lassen sich vielseitiger einsetzen als lediglich einen Container mit einem Elementtyp zu parametrisieren. So verwendet man sie ausgiebig für die Parametrisierung sowohl von Typen als auch von Algorithmen in der Standardbibliothek (§9.6, §10.6). Zum Beispiel können wir eine Funktion schreiben, die die Summe der Elementwerte eines beliebigen Containers berechnet:

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{
    for (auto x : c)
        v+=x;
    return v;
}
```

Das Template-Argument **Value** und das Funktionsargument **v** erlauben dem Aufrufer, den Typ und den Anfangswert des Akkumulators (der Variablen, in der die Summe aufläuft) festzulegen:

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc)
{
    int x = sum(vi,0);           // Summe eines int-Vektors (ints addieren)
    double d = sum(vi,0.0);      // Summe eines int-Vektors (doubles addieren)
    double dd = sum(ld,0.0);     // Summe einer Liste von doubles
    auto z = sum(vc,complex<double>{}); // Summe eines Vektors von complex<double>
                                   // Anfangswert ist {0.0,0.0}
}
```

Durch das Addieren von **int**-Werten in einem **double**-Typ lässt sich eine Zahl größer als die größte **int**-Zahl elegant verarbeiten. Beachten Sie, wie die Typen der Template-Argumente für **sum<T,V>** aus den Funktionsargumenten hergeleitet werden. Erfreulicherweise brauchen wir diese Typen nicht explizit zu spezifizieren.

Diese **sum()**-Funktion ist eine vereinfachte Version der Funktion **accumulate()** aus der Standardbibliothek (§12.3).

■ 5.4 Konzepte und generische Programmierung

Wofür sind Templates vorgesehen? Oder anders ausgedrückt: Welche Programmieretechniken sind effektiv, wenn man Templates verwendet? Templates bieten:

- die Möglichkeit, Typen (wie auch Werte und Templates) als Argumente zu übergeben, ohne Informationen zu verlieren. Das bedeutet ausgezeichnete Möglichkeiten für Inlining, wovon aktuelle Implementierungen stark profitieren.
- verzögerte Typprüfung (die erst zur Instanziierungszeit erfolgt). Dies bietet Gelegenheit, Informationen aus verschiedenen Kontexten miteinander zu verweben.
- die Fähigkeit, konstante Werte als Argumente zu übergeben. Daraus ergibt sich die Möglichkeit, Berechnungen zur Übersetzungszeit anzustellen.

Mit anderen Worten bieten Templates einen leistungsfähigen Mechanismus für Berechnungen und Typmanipulationen zur Übersetzungszeit, was zu sehr kompaktem und effizientem Code führen kann. Denken Sie daran, dass Typen (Klassen) sowohl Code als auch Werte enthalten können.

Templates verwendet man in erster Linie und am häufigsten, um *generische Programmierung* zu unterstützen, d. h. Programmierung, die sich auf das Design, die Implementierung

und die Verwendung von allgemeinen Algorithmen konzentriert. Hier bedeutet „allgemein“, dass ein Algorithmus entworfen werden kann, der ein breites Spektrum von Typen akzeptiert, sofern diese die Anforderungen des Algorithmus an seine Argumente erfüllen. Das Template ist in C++ das Hauptinstrument für die generische Programmierung. Templates bieten parametrische Polymorphie (zur Übersetzungszeit).

Sehen Sie sich dazu die Funktion `sum()` von §5.3 an. Man kann sie für jede Datenstruktur aufrufen, die die Funktionen `begin()` und `end()` implementiert, damit die bereichsbasierte `for`-Anweisung funktioniert. Mit derartigen Strukturen arbeiten die Klassen `vector`, `list` und `map` der Standardbibliothek. Darüber hinaus wird der Elementtyp der Datenstruktur nur durch seine Verwendung beschränkt: Es muss sich um einen Typ handeln, den wir zum `Value`-Argument addieren können. Der Algorithmus `sum()` ist gewissermaßen in zweierlei Hinsicht generisch: in Bezug auf den Typ der für das Speichern der Elemente verwendeten Datenstruktur („Container“) und den Typ der Elemente.

Somit verlangt `sum()`, dass ihr erstes Template-Argument ein Art von Container ist und ihr zweites Template-Argument eine Art von Zahl. Derartige Anforderungen bezeichnet man als *Konzepte*. Leider lassen sich Konzepte nicht direkt in C++11 ausdrücken. Wir können lediglich sagen, dass das Template-Argument für `sum()` ein Typ sein muss. Es gibt zwar Techniken, um Konzepte zu überprüfen, und Vorschläge für eine direkte Sprachunterstützung von Konzepten ([Stroustrup, 2013], [Sutton, 2011]), doch gingen beide Themen über den Rahmen dieses Buchs hinaus.

Gute und nützliche Konzepte sind von elementarer Bedeutung. Zudem werden sie eher entdeckt als entworfen. Beispiele sind Ganzzahlen und Gleitkommazahlen (wie sie selbst in klassischem C definiert sind), allgemeinere mathematische Konzepte wie Feld und Vektorraum sowie Container. Diese repräsentieren die fundamentalen Konzepte eines Einsatzgebiets. Es kann eine Herausforderung sein, sie zu identifizieren und bis zum erforderlichen Grad für effektive generische Programmierung zu formalisieren.

Das Prinzip soll am Konzept *Regulär* veranschaulicht werden. Ein Typ ist regulär, wenn er sich ähnlich wie ein `int` oder ein `vector` verhält. Ein Objekt eines regulären Typs

- kann standardmäßig konstruiert werden,
- lässt sich mit einem Konstruktor oder einer Zuweisung kopieren (mit der üblichen Kopiersemantik, die zwei voneinander unabhängige und auf Gleichheit vergleichbare Objekte liefert),
- kann mit `==` und `!=` verglichen werden und
- ist frei von technischen Problemen infolge übermäßig cleverer Programmiertricks.

Ein `string` ist ein weiteres Beispiel für einen regulären Typ. Wie `int` ist `string` auch *geordnet*. Das bedeutet, dass sich zwei Strings mit den Operatoren `<`, `<=`, `>` und `>=` mit der geeigneten Semantik vergleichen lassen. Konzepte sind mehr als eine syntaktische Vorstellung. Sie verkörpern vielmehr die fundamentale Semantik. Zum Beispiel sollten Sie `+` nicht als Divisionsoperator definieren; dies widerspräche den Anforderungen an sinnvolle arithmetische Operationen.

■ 5.5 Funktionsobjekte

Eine besonders nützliche Art von Template ist das *Funktionsobjekt* (auch als *Funktor* bezeichnet), mit dem man Objekte definiert, die sich wie Funktionen aufrufen lassen. Zum Beispiel:

```
template<typename T>
class Less_than {
    const T val;                // Wert, mit dem verglichen werden soll
public:
    Less_than(const T& v) : val(v) { }
    bool operator()(const T& x) const { return x < val; } // Operator aufrufen
};
```

Die Funktion namens **operator()** implementiert den „Funktionsaufrufs-“, „Aufrufs-“ oder „Anwendungs-“ Operator **()**.

Wir können benannte Variablen des Typs **Less_than** für einen bestimmten Argumenttyp definieren:

```
Less_than<int> lti {42};        // lti(i) vergleicht i
                                // mit 42 mithilfe von < (i<42)
Less_than<string> lts {"Backus"}; // lts(s) vergleicht s
                                // mit "Backus" mithilfe von < (s<"Backus")
```

Ein solches Objekt können wir wie eine Funktion aufrufen:

```
void fct(int n, const string & s)
{
    bool b1 = lti(n);    // true, wenn n<42
    bool b2 = lts(s);    // true, wenn s<"Backus"
    // ...
}
```

Derartige Funktionsobjekte sind als Argumente für Algorithmen gebräuchlich. Zum Beispiel können wir das Auftreten von Werten zählen, für die ein Prädikat **true** zurückgibt:

```
template<typename C, typename P>
int count(const C& c, P pred)
{
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

Ein *Prädikat* ist etwas, das wir aufrufen können, um **true** oder **false** zurückzugeben. Zum Beispiel:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "Anzahl der Werte kleiner als " << x
         << ": " << count(vec, Less_than<int>{x})
         << '\n';
    cout << "Anzahl der Werte kleiner als " << s
```



```

    << ": " << count(lst, Less_than<string>{s})
    << '\n';
}

```

Hier konstruiert `Less_than<int>{x}` ein Objekt, für das der Aufrufoperator den `int` namens `x` vergleicht, und `Less_than<string>{s}` konstruiert ein Objekt, das den `string` namens `s` vergleicht. Der Reiz dieser Funktionsobjekte liegt darin, dass sie den Wert, mit dem der Vergleich stattfinden soll, mit sich führen. Wir brauchen keine separate Funktion für jeden Wert (und jeden Typ) zu schreiben und wir kommen ohne hässliche globale Variablen aus, um Werte zwischenzuspeichern. Außerdem lassen sich einfache Funktionsobjekte wie etwa `Less_than` problemlos als Inline-Funktion realisieren, sodass ein Aufruf von `Less_than` weitaus effizienter ist als ein indirekter Funktionsaufruf. Die Möglichkeit, Daten mitzuführen, plus ihre Effizienz machen Funktionsobjekte besonders nützlich als Argumente in Algorithmen.

Funktionsobjekte, mit denen man die Bedeutung von wichtigen Operationen eines allgemeinen Algorithmus spezifiziert (wie zum Beispiel `Less_than` für `count()`) heißen auch *Richtlinienobjekte* (engl. policy objects).

Das Funktionsobjekt `Less_than` ist getrennt von seiner Verwendung zu definieren. Das mag unkomfortabel aussehen. Deshalb gibt es auch ein Konzept, um Funktionsobjekte implizit zu generieren:

```

void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
    cout << "Anzahl der Werte kleiner als " << x
        << ": " << count(vec, [&](int a){ return a<x; })
        << '\n';
    cout << "Anzahl der Werte kleiner als " << s
        << ": " << count(lst, [&](const string& a){ return a<s; })
        << '\n';
}

```

Bei der Notation `[&](int a){ return a<x; }` handelt es sich um einen sogenannten *Lambda-Ausdruck*. Er generiert ein Funktionsobjekt exakt so wie `Less_than<int>{x}`. Die mit `[&]` bezeichnete *Erfassungsliste* (engl. capture list) gibt an, dass auf die verwendeten lokalen Namen (wie zum Beispiel `x`) über Referenzen zugegriffen wird. Hätten wir nur `x` „erfassen“ wollen, könnten wir das mit `[&x]` ausdrücken. Um dem generierten Objekt eine Kopie von `x` zu geben, schreiben wir `[=x]`. Des Weiteren bedeutet `[]`, nichts zu erfassen, `[&]`, alle als Referenz verwendeten lokalen Namen zu erfassen, und `[=]`, alle lokalen Variablen zu erfassen und als Wert zu verwenden.

Lambda-Ausdrücke können komfortabel und prägnant, aber auch undurchsichtig sein. Für nichttriviale Aktionen (sagen wir, mehr als einen einfachen Ausdruck) ziehe ich es vor, die Operation zu benennen, um damit ihren Zweck zu verdeutlichen und sie an mehreren Stellen in einem Programm verwendbar zu machen.

In §4.5.4 haben wir festgestellt, wie lästig es ist, zu viele Funktionen schreiben zu müssen, um Operationen mit `vector`-Zeigerelementen und `unique_ptr`-Objekten auszuführen, wie zum Beispiel `draw_all()` und `rotate_all()`. Funktionsobjekte (insbesondere Lambda-Ausdrücke) kommen uns hier zu Hilfe: Damit können wir das Durchlaufen des Containers von der Spezifikation, was mit jedem Element zu tun ist, trennen.

Zuerst brauchen wir eine Funktion, die eine Operation auf jedes Objekt anwendet, auf das die Elemente eines Containers von Zeigern zeigen:

```
template<typename C, typename Oper>
void for_all(C& c, Oper op)    // annehmen, dass C ein Container von Zeigern ist
{
    for (auto& x : c)
        op(*x);               // op() eine Referenz auf jedes Element übergeben, auf
                               // das gezeigt wird
}
```

Jetzt können wir eine Version von **user()** aus §4.5 schreiben, ohne einen Satz aller **_all**-Funktionen schreiben zu müssen:

```
void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    for_all(v, [](Shape& s){ s.draw(); });    // draw_all()
    for_all(v, [](Shape& s){ s.rotate(45); }); // rotate_all(45)
}
```

Ich übergebe eine Referenz auf **Shape** an einen Lambda-Ausdruck, sodass sich der Lambda-Ausdruck nicht darum kümmern muss, wie die Objekte im Container gespeichert werden. Insbesondere funktionieren diese **for_all()**-Aufrufe immer noch, wenn ich **v** in einen **vector<Shape*>** ändere.

■ 5.6 Variadische Templates

Für ein Template lässt sich definieren, dass es eine beliebige Anzahl von Argumenten beliebiger Typen übernimmt. Eine derartiges Template heißt *variadisches Template*. Zum Beispiel:

```
void f() {}    // nichts tun

template<typename T, typename... Tail>
void f(T head, Tail... tail)
{
    g(head);    // etwas mit head unternehmen
    f(tail...); // erneut mit tail probieren
}
```

Wichtig bei der Implementierung eines variadischen Templates ist es, dass Sie das erste Argument von den übrigen trennen können, wenn Sie ihm eine Liste von Argumenten übergeben. Hier unternehmen wir etwas mit dem ersten Argument (dem **head**) und rufen dann rekursiv **f()** mit den übrigen Argumenten (dem **tail**) auf. Die Ellipse (...) steht für „den Rest“ einer Liste. Natürlich wird schließlich **tail** leer und wir brauchen eine eigene Funktion, die dies verarbeitet.

Diese Funktion **f()** können wir wie folgt aufrufen:

```
int main()
{
    cout << "erstes: ";
    f(1,2.2,"hello");

    cout << "\nzweites: ";
    f(0.2,'c',"yuck!",0,1,2);
    cout << "\n";
}
```

Dies ruft **f(1,2.2,"hello")** auf, was wiederum **f(2.2,"hello")** aufruft, was **f("hello")** aufruft, was **f()** aufruft. Was bewirkt nun der Aufruf **g(head)**? Offensichtlich wird er in einem echten Programm das tun, was wir für jedes Argument vorgesehen haben. Zum Beispiel können wir ihn veranlassen, sein Argument (hier **head**) in die Ausgabe zu schreiben:

```
template<typename T>
void g(T x)
{
    cout << x << " ";
}
```

Damit lautet die Ausgabe:

```
erstes: 1 2.2 hello
zweites: 0.2 c yuck! 0 1 2
```

Es sieht so aus, als ob **f()** eine einfache Variante von **printf()** ist, die beliebige Listen von Werten ausgibt – implementiert in drei Zeilen Code plus die umgebenden Deklarationen.

Die Stärke von variadischen Templates ist, dass sie beliebige Argumente akzeptieren, die Sie übergeben möchten. Ihre Schwäche liegt darin, dass die Typprüfung der Schnittstelle ein möglicherweise aufwendiges Template-Programm ist.

Aufgrund der Flexibilität von variadischen Templates greift die Standardbibliothek ausgiebig darauf zurück.

■ 5.7 Alias

Überraschend oft ist es nützlich, ein Synonym für einen Typ oder ein Template einzuführen. Zum Beispiel enthält der Standard-Header **<cstdint>** eine Definition des Alias **size_t**:

```
using size_t = unsigned int;
```

Der tatsächliche Typ namens **size_t** ist implementierungsabhängig, sodass in einer anderen Implementierung **size_t** ein **unsigned long** sein kann. Mit dem Alias **size_t** ist der Programmierer in der Lage, portablen Code zu schreiben.

Für einen parametrisierten Typ ist es durchaus üblich, einen Alias für Typen in Bezug auf ihre Template-Argumente bereitzustellen. Zum Beispiel:

```
template<typename T>
class Vector {
public:
    using value_type = T;
    // ...
};
```

In der Tat stellt jeder Container der Standardbibliothek **value_type** als Name seines Werttyps bereit (Kapitel 9). Damit können wir Code schreiben, der für jeden Container funktioniert, der sich an diese Konvention hält. Zum Beispiel:

```
template<typename C>
using Element_type = typename C::value_type; // Typ der Elemente von C

template<typename Container>
void algo(Container& c)
{
    Vector<Element_type<Container>> vec;    // Ergebnisse hier speichern
    // ...
}
```

Mit Aliasing lässt sich ein neues Template definieren, indem man einige oder alle Template-Argumente bindet. Zum Beispiel:

```
template<typename Key, typename Value>
class Map {
    // ...
};

template<typename Value>
using String_map = Map<string, Value>;

String_map<int> m;    // m ist ein Map<string, int>
```

■ 5.8 Modell der Template-Kompilierung

Die für Templates bereitgestellte Typprüfung untersucht die Verwendung von Argumenten in der Template-Definition und nicht in Bezug auf eine explizite Schnittstelle (in einer Template-Deklaration). Dies bietet eine Übersetzungszeitvariante, die man auch als *Duck-Typing* bezeichnet („Wenn es wie eine Ente watschelt und wie eine Ente schnattert, dann ist es eine Ente“). Etwas technischer ausgedrückt: Wir operieren auf Werten und die Anwesenheit und Bedeutung einer Operation hängt allein von ihren Operandenwerten ab. Dies unterscheidet sich von der alternativen Sichtweise, dass Objekte über Typen verfügen, die die Anwesenheit und Bedeutung von Operationen bestimmen. Werte „leben“ in Objekten. Auf diese Weise arbeiten Objekte (z.B. Variablen) in C++, und nur Werte, die den Anforderungen eines Objekts genügen, lassen sich im Objekt unterbringen. Bei den Abläufen zur Übersetzungszeit mithilfe von Templates sind keine Objekte beteiligt, nur Werte.

Für die Praxis bedeutet das, dass sich die Definition eines Templates (und nicht nur seine Deklaration) im Gültigkeitsbereich befinden muss. Zum Beispiel enthält der Standard-Header **<vector>** die Definition von **vector**. Ein unglücklicher Nebeneffekt ist, dass sich ein

Typfehler erst sehr spät im Übersetzungsvorgang bemerkbar macht, was gegebenenfalls zu spektakulären Fehlermeldungen führt, weil der Compiler das Problem anhand von Informationen ermittelt, die von verschiedenen Stellen im Programm stammen.

■ 5.9 Ratschläge

1. Der Stoff in diesem Kapitel entspricht in etwa dem, was die Kapitel 20 bis 29 von [Stroustrup, 2013] wesentlich detaillierter beschreiben.
2. Verwenden Sie Templates, um Algorithmen darzustellen, die sich auf viele Argumenttypen anwenden lassen; §5.1.
3. Verwenden Sie Templates, um Container darzustellen; §5.2.
4. Verwenden Sie Templates, um das Abstraktionsniveau des Codes anzuheben; §5.2.
5. Wenn Sie ein Template definieren, sollten Sie zuerst eine Nicht-Template-Version entwerfen und debuggen. Später fügen Sie Parameter hinzu, um die getestete Version zu verallgemeinern.
6. Templates sind zwar typsicher, doch findet die Überprüfung zu spät statt; §5.4.
7. Ein Template kann Argumenttypen ohne Informationsverlust übergeben.
8. Verwenden Sie Funktions-Templates, um Argumenttypen von Klassen-Templates her-zuleiten; §5.3.
9. Templates bieten einen allgemeinen Mechanismus für die Programmierung zur Über-setzungszeit; §5.4.
10. Wenn Sie ein Template entwerfen, achten Sie sorgfältig auf die Konzepte (Anforderun-gen), die für seine Template-Argumente angenommen werden; §5.4.
11. Verwenden Sie Konzepte als Entwurfswerkzeug; §5.4.
12. Verwenden Sie Funktionsobjekte als Argumente für Algorithmen; §5.5.
13. Verwenden Sie einen Lambda-Ausdruck, wenn Sie ein einfaches Funktionsobjekt an nur einer Stelle benötigen; §5.5.
14. Eine virtuelle Funktion kann keine Template-Funktion sein.
15. Verwenden Sie Template-Alias, um die Notation zu vereinfachen und Implementie-rungsdetails zu verbergen; §5.7.
16. Verwenden Sie variadische Templates für eine Funktion, die eine variable Anzahl von Argumenten mit einer breiten Palette von Typen übernehmen soll; §5.6.
17. Verwenden Sie für homogene Argumentlisten keine variadischen Templates (bevorzu-gen Sie hierfür Initialisierungslisten); §5.6.
18. Achten Sie bei Verwendung eines Templates darauf, dass sich seine Definition (und nicht nur seine Deklaration) im Gültigkeitsbereich befindet; §5.8.
19. Templates bieten „Duck-Typing“ zur Übersetzungszeit; §5.8.
20. Es gibt keine getrennte Übersetzung von Templates: Binden Sie Template-Definitionen per `#include` in jede Übersetzungseinheit ein, die sie verwendet.

Index

Symbole

// (Kommentar) 2
.
(Punktoperator) 19
... (Ellipse) 70
[] (Indexoperator) 80
{ } (Gruppierung) 2
+= (Verkettung) 79
=delete 59

A

Ableitung 45
abs() 143
Absolutwert 143
abstrakt
– Klassen 42
– Typen 41
Abstraktionsmechanismen 17
accumulate() 144
acos() 143
Adapter
– bind() 134
– function 136
– Funktions- 134
– regex_iterator 88
adjacent_difference() 145
Algorithmen 115
– Container 123
– Kopieren 123
– Sequenzen 115
– sort() 138
– Sortieren 123
– Standard- 122
– STL 124

– Überblick 122
– Verschieben 123
– Zusammenführen 123
Alias 71
– string 81
Anweisungen
– bereichsbasierte for- 11
– Deklarationen 5
– throw 30
– while 13
Argumente
– Kopieren 12
– Templates 65
– übergeben 153
– Wert- 65
Arithmetik 5
– Operationen 145
– Übersetzungszeit 171
– Vektoren 148
– Zeiger- 177
Arkuskosinus 143
Arkussinus 144
Arkustangens 144
ARM (The Annotated C++ Reference Manual)
167
array 129, 130
– Initialisierungslisten 130
Arrays 10, 65
– assoziative 110, 112
– Container 129
– Grenzen 10
– Größe, feste 112, 130
– Länge, variable 175
– nullterminierte 80
– Schreiben über Zeiger 120

- string 36
- vector 36
- Vektoren 18
- vs. array 131
- asin() 144
- Assertionen 32
- async() 160
- at() 107
- atan() 144
- Aufzählungen 22
 - Gültigkeitsbereiche 22
- Ausdrücke
 - konstante 10
 - Operanden 7
 - reguläre 82
- Ausgabe 92
 - Ganzzahlen 97
- Auslassungszeichen 70
- Ausnahmen 29
 - bad_alloc 31
 - bad_cast 50
 - catch 30
 - erneut auslösen 32
 - Fehler 58
 - length_error 31
 - out_of_range 30, 107
 - throw 30
 - try 30
 - veraltete 172
- Auswertung, partielle 134
- auto 8

B

- Backslashes 3
 - reguläre Ausdrücke 87
 - Stringlitterale 3, 82
- bad_alloc 31
- bad_cast 50
- basic_string 129
- Basisklassen 43
- Bedingungsvariablen 156
- begin() 108
- Beinahe-Container 129
- Benutzeroberflächen, grafische (GUI) 75
- Bereiche
 - Fehler 106
 - halboffene 30

- Vektoren 29, 104
- Zugriff auf Elemente außerhalb 29
- bereichsbasierte for-Anweisungen 11
- Bereichsüberprüfung 38, 106
 - Iteratoren 119
- Bibliotheken 75
- bind() 134, 146
- Binder 134
- bitset 129, 131
- Blöcke 8
 - try 30

C

- C++
 - C-Makros 178
 - Nebenläufigkeit 151
 - Programmierung, generische 66
 - Ressourcen 126
 - Schlüsselwörter 177
- C++11
 - Erweiterungen 169
 - Features 169
- capture list (Erfassungsliste) 69
- Casts 172
- catch 30
- ceil() 143
- cerr 92, 107
- <chrono> 156
- <cmath> 143
- Compiler 1
- complex 37
- <condition_variable> 156
- const 9, 37
- constexpr 9
- Container 38, 103
 - Algorithmen 123
 - array 130
 - Beinahe- 129
 - bitset 131
 - Elemente 106
 - Initialisierung 40
 - kopieren 52
 - Ordnungsfunktionen 110
 - spezialisierte 129
 - Überblick 111
 - vector 103
 - verschieben 53

cos() 143
cosh() 144
cout 92
C-Strings 13
Currying 134

D

Data Races 152
Daten
– Funktionsobjekte 69
– gemeinsam nutzen 154
– Iteratoren 118
– Operationen 19
– Strukturen 17
– Zeit- 134
Datum und Uhrzeit 134
Deadlocks 155
Definitionen 25
Deklarationen 5, 25
– einbinden 2
– include 2
– Operatoren 12
Deklarationsoperatoren 12
deprecated (veraltet) 171
Destruktoren 38
dictionary (Wörterbuch) 110
Direktiven 28
distribution 146
Domänenfehler 144
Duck-Typing 72
dynamic_cast 50
– instanceof 50
– Typumwandlung 50, 173
dynamischer Speicher 18

E

E/A-Streams 91
ECMAScript 84
EDOM 144
Ein-/Ausgabe
– benutzerdefinierte Typen 96
– Formatierung 97
einbinden 2
Eingabe 93
– push_back() 40
einschränkende Konvertierungen 7

Elemente
– allozieren 40
– Arrays 10
– außerhalb eines Bereichs 29
– Container 38, 106
– Indizierung 104
– Iteratoren 108
– kopieren 53
– push_back() 40
– Summe bilden 159
– Zeiger 103
Ellipse 70
end() 108
engine 146
enum class 22
Enumerationen (Aufzählungen) 22
equal_range() 133
ERANGE 144
Ereignisse
– Simulationen 165
– Threads 156
– Warten auf 156
Erfassungslisten 69
Estd 124
exp() 144
explicit 57
Exponentialfunktion 144
export 172, 177
extern 178

F

Features
– deprecated (veraltete) 171
– inkompatible 175
– Kernsprache 1
– Sprach- 169
Fehler
– Ausnahmen 58
– Bereichs- 106, 144
– Domänen- 144
– EDOM 144
– ERANGE 144
– Formatierung 105
– Nebenläufigkeit 152
– Ressourcenlecks 50
– Rückgabewerte 2
– Standardfehlerausgabe 107

- Fehlerbehandlung 29
- floor() 143
- for
 - bereichsbasiert 11, 64
 - Gültigkeitsbereich 40
- Formate
 - allgemeine 98
 - feste 98
 - Genauigkeit 98
 - wissenschaftliche 98
- Formatierung
 - Ein-/Ausgabe 97
 - Manipulatoren 97
- forward iterators 138
- Freispeicher 18
 - Gültigkeitsbereiche 18
 - Konstruktoren 39
- fstream 98
- <fstream> 98
- function 136
- fundamentale Typen 5
- Funktionen
 - Argumente 4
 - Binden überladener 135
 - <cmath> 143
 - const 37
 - Definitionen 25
 - Deklarationen 3
 - inline 37
 - komplexe Zahlen 145
 - Konstruktoren 20
 - main() 2
 - make_pair() 133
 - rein virtuelle 42
 - spezielle mathematische 144
 - Tabelle für virtuelle 44
 - Templates 65
 - Typ- 137
 - überladen 5
 - virtuelle 42, 44
- Funktionsadapter 134
 - mem_fn() 135
- Funktionsdeklarationen 3
- Funktionsobjekte 68
 - bind() 135, 146
- Funktionstabellen, virtuelle 44
- Funktoren 68
- future 158

G

- Ganzzahlen
 - Ausgabe 97
 - Divisionsrest 6
 - Genauigkeit 98
 - größte 41, 143
 - Inkrement 11
 - kleinste 143
 - lesen 93
 - Typnamen 171
 - Verteilung 146
- Genauigkeit 98
- generische Programmierung 66
- getline() 94
- Glockenkurve 146
- grafische Benutzeroberflächen 75
- Graphical User Interface (GUI) 75
- greedy match (gierige Übereinstimmung) 88
- Größen
 - Arrays 112, 130
 - Objekte 137
 - Typen 6
 - Vektoren 104
- Gruppen (reguläre Ausdrücke) 87
- GUI (Graphical User Interfaces) 75
- Gültigkeitsbereiche
 - Aufzählungen 22
 - Definition 8
 - Enumeratoren 23
 - for 40
 - Namen, lokale 8
 - Objekte im Freispeicher 18
 - Objekte verschieben 57
 - Ressourcen 58
 - Templates 72
 - Threads 156
 - using 28
 - while 40

H

- Hashfunktionen 111
- Hashtabellen 112
- Header-Dateien 26
- Heap 18
- Hello, World! (Beispiel) 2

Hierarchien

- Klassen- 45
- Navigation 50
- Vorzüge 48

I

ifstream 98

Implementierungsvererbung 48

include (einbinden) 2

Indizierung

- Elemente 104
- Strings 80

Initialisierung

- Container 40
- Listen 7

Initialisierungslisten 130

- Konstruktoren 40

inline 37, 177

inner_product() 144

instanceof (dynamic_cast) 50

Invarianten 30

- Ressourcen-Handles 52

iostream

- Formatierung 97
- Status 95

istream 91, 93

istream_iterator 120

istreamstream 99

Iteratoren 108, 116

- istream_iterator 120
 - list 119
 - mit wahlfreiem Zugriff 138
 - ostream_iterator 120
 - pair 132
 - regex_iterator 88
 - reguläre Ausdrücke 88
 - Sequenzen 115
 - Streams 120
 - Typen 119
 - vector 119
 - Vorwärts- 138
- iterator_traits 138

J

JavaScript 84

K

key (Schlüssel) 110

Klammern

- eckige 10
- Funktionsnamen 4
- geschweifte 2
- Größe von Vektoren 104
- Gruppen 87
- spitze 64, 100
- Teilmuster 84

Klassen 19, 35

- abgeleitete 43
- abstrakte 42
- Aufzählungs- 22
- Basis- 43
- konkrete 36
- Sub- 43
- Super- 43
- Vec 107
- Vererbung 43

Klassenhierarchien

- Ableitung 45
- Funktionen, virtuelle 106
- Navigation 173
- Vorzüge 48

Klasseninvarianten 31

Kommentare 2

Kompatibilität

- C/C++ 173
- Konvertierungen, einschränkende 7
- Probleme 175

Kompilierung

- Header einbinden 77
- reguläre Ausdrücke 83
- separate 26
- Template- 72

komplexe Zahlen 145

konkrete Typen 41

Konstanten 9

- Zeichen 93

Konstruktoren 20

- Freispeicher 39
- Initialisierungslisten 40
- Kopier- 52
- Optimierung 56
- Standard- 37
- Verschiebe- 54

Konvertierungen
– auto 8
– dynamic_cast 50
– einschränkende 7, 173
– explicit 57
– übliche arithmetische 7
– Zeit 134
Konzepte 67
– Typen, reguläre 67
Kopieren
– Algorithmen 123
– Argumente 12
– Container 52
– memberweises 52
– Objekte 51
– Vector-Objekte 106
Kopierkonstruktoren 52
Kopierzuweisungen 52
Kosinus 143
Kosinus Hyperbolicus 144

L

Lambda-Ausdrücke 69
lazy match (faule Übereinstimmung) 88
Lebensdauer 8
– shared_ptr 127
– steuern 57
– unique_ptr 127
– Vector-Elemente 125
length_error 31
<limits> 137, 148
Linker 1
list 43, 108
– Iteratoren 119
Listen
– Argumente 70
– Deque 112
– doppelt verkettete 108, 111
– einfach verkettete 111
– Initialisierung 7, 40
– list 43
locks (Sperrn) 125, 152
log() 144
log10() 144
Logarithmus
– dekadischer 144
– natürlicher 144

M

main() 2
make_pair() 133
make_shared() 128
Makros
– C- 178
– C99- 170
– Ersetzung 176
– Parametrisierung 165
Manipulatoren 97
map 109
Maps 112
mathematische Standardfunktionen 143
Member 19
mem_fn() 135
Mengen 112
Metaprogrammierung 138
Metazeichen 84
Modularität 25
Multimaps 112
Multimengen 112
Mustermaschine 85
mutal exclusion (wechselseitiger Ausschluss)
155
mutex 155
<mutex> 155

N

Namen
– Ganzzahltypen 171
– globale 9
– lokale 8
Namespaces 28
– Estd 124
– globale 9
– placeholders 135
– std 77
– std::chrono 134
– using 28
Navigation (Hierarchien) 50, 173
Nebenläufigkeit 151
Newline 3
nicht gefunden 117
non-greedy match (zurückhaltende Übereinstimmung) 88
normal_distribution 146

Notation

- Bereiche, halboffene 30
- Lambda-Ausdrücke 69
- reguläre Ausdrücke 84
- Standardcontainer 112

NULL 13

nullptr 12

Nullzeiger 12

<numeric> 144

numeric_limits 137

O

Objektdateien 1

Objekte 5

- benannte 5
- Größe 137
- kopieren 51
- sizeof 137
- verschieben 51

ofstream 98

Operanden

- Ausdrücke 7
- Ergebnistyp 6
- rechtsseitige 14, 93
- Templates 72
- void* 177

Operationen

- Arithmetik 145
- Daten 19
- komplexe Zahlen 145
- Status 95
- unterdrücken 59
- wichtige 55

Operatoren

- Deklarations- 12
- dynamic_cast 50
- Punkt- 19
- sizeof 6
- static_cast 41
- überladen 38

Optimierung

- Berechnungen, numerische 148
- bitset 131
- Konstruktoren 56
- Strings, kurze 81
- Suche 110
- valarray 148

Ordnungsfunktionen 110

ostream 91

ostream_iterator 120

ostreamstring 99

out_of_range 30, 107

override (überschreiben) 43

P

packaged_task 159

pair 129, 132

partial_sum() 145

partielle Auswertung 134

Pattern Matcher (Mustermaschine) 85

placeholders 135

Platzhalter 135

policy objects (Richtlinienobjekte) 69

polymorphe Typen 42

Polymorphie

- parametrische 66
- Templates 66

Portabilität 1

- reguläre Ausdrücke 86
- Zeichenklassen 86

Prädikate 68, 122

precision() 98

Probleme

- Bereichsfehler 106
- Kompatibilität 175
- Konvertierungen, einschränkende 7
- Nebenläufigkeit 154
- new 127
- numerische 149
- Performance 55
- Sonderzeichen 87
- Stil 175
- Variablen, nichtinitialisierte 20

Programme 1

- ausführbare 1
- main() 2
- portable 1

Programmierung

- generische 5, 35, 66
- nebenläufige 76, 152
- objektorientierte 35
- parallele 76
- prozedurale 1
- sperrenfreie 151

- System- 2, 163
- Systeme, eingebettete 164
- Template-Meta- 76, 138
- Templates 164
- promise 158
- Punktooperator 19
- push_back() 40, 104

Q

- Quadratwurzel 143
- Quelltext 1
- queue 157

R

- Race Conditions 157
- RAII (Resource Acquisition Is Initialization) 58
- <random> 146
- random-access iterators 138
- Referenzen 10
- Regel der längsten Übereinstimmung 85
- <regex> 82
- regex_iterator 82, 88
- regex_match() 82
- regex_replace() 82
- regex_search() 82
- regex_token_iterator 82
- reguläre Ausdrücke 82
 - Backslashes 87
 - ECMA 84
 - Gruppen 87
 - Iteratoren 88
 - JavaScript 84
 - Metazeichen 84
 - Notation 84
 - Portabilität 86
 - regex_iterator 88
 - smatch 83
 - suchen 83
 - Wiederholungen 84
 - Zeichenklassen 85
 - Zustandsmaschinen 83
- rein virtuelle Funktionen 42
- replace() 80
- resource retention (Ressourcenbindung) 58

- Ressourcen
 - belegen 49
 - Bindung 58
 - freigeben 49
 - Handles 36, 52, 160
 - Lecks vermeiden 50, 58
 - Sicherheit 58
 - System- 161
 - Verwaltung 57, 125
- Ressourcenbelegung ist Initialisierung 30, 40, 58, 126
- Richtlinienobjekte 69
- rohe Stringlitterale 82
- Rot-Schwarz-Bäume 109
- Rückgabewerte 38
 - Fehler 2
 - Funktionen 3
 - kopieren 156
 - nicht gefunden 117
 - System 2
 - Tasks 159
 - Übertragen aus Funktion 54
- R-Wert-Referenzen 54, 175

S

- Schleifen 13
 - while 13
- Schlüssel 110
- Schlüsselwörter 177
 - inline 37
 - virtual 42
- Schnittstellen 19
 - Deklarationen 25
- Schnittstellenvererbung 48
- scope (Gültigkeitsbereich) 8
- Sequenzen
 - Algorithmen 115
 - Ende 117
 - halboffene 115
 - nicht gefunden 117
 - zusammenführen 123
- shared pointers (gemeinsame Zeiger) 58
- shared_ptr 126
- Short String Optimization 81
- sin() 143
- sinh() 144
- Sinus 143

- Sinus Hyperbolicus 144
- sizeof 6, 137
- smatch 83
- Sonderzeichen 3
- sort() 116, 138
 - Container 124
- Sortieren 109
 - Algorithmen 123
 - Vektoren 138
- Speicher
 - dynamischer 18
 - Frei- 18
 - Heap 18
- sqrt() 143
- Standardbibliothek 167
 - C++11-Ergänzungen 170
 - getline() 94
 - Header 77
 - Hilfskomponenten 125
 - istream 93
 - Komponenten 76
 - list 43, 108
 - map 109
 - Namespaces 77
 - ostream 92
 - Suchbaum 109
 - Überblick 76
 - unordered_map 111
- Standardfunktionen 143
- Standardkonstruktoren 37
- Standards
 - ECMA 84
 - ISO-C++- 167
- static_assert 32
- static_cast 41
- Status
 - Fehler 120
 - Flags 131
 - istream 95
- std 77
- std::chrono 134
- STL 76
 - Algorithmen 124
 - Beinahe-Container 129
 - Container 113, 130
 - Framework 164
 - Programmierung, generische 76
 - Standardbibliothek 167

- Streams
 - Ausgabe 92
 - Datei- 98
 - Ein-/Ausgabe 91
 - Eingabe 93
 - Formatierung 97
 - <fstream> 98
 - Iteratoren 120
 - ostream 91
 - reguläre Ausdrücke 88
 - Status 95
 - Strings 99
- Stringliterale 3
 - Initialisierung 172
 - reguläre Ausdrücke 87
 - rohe 82
 - vergleichen 80
 - Zuweisung 172
- Strings 79
 - C- 13
 - Optimierung für kurze 81
 - replace() 80
 - Streams 99
 - substr() 80
 - verketten 79
 - Zeichensätze 81
- stringstream 99
- Strukturen 17
- Subklassen 43
- substr() 80
- Suchen 109
 - nicht gefunden 124
 - Records 133
 - reguläre Ausdrücke 83
 - Rot-Schwarz-Bäume 109
 - Strings 82
- Superklassen 43
- Systemprogrammierung 2, 163
- Systemressourcen 161

T

- Tag-Dispatching 139
- tagged Unions 22
- tan() 143
- Tangens 143
- Tangens Hyperbolicus 144
- tanh() 144

- Tasks 152
 - asynchrone 161
 - Kommunikation 158
- Template-Metaprogrammierung 138
- Templates 63
 - Argumente 65
 - Duck-Typing 72
 - Funktionen 65
 - Funktionsobjekte 68
 - Gültigkeitsbereiche 72
 - Programmierung, generische 66
 - variadische 70, 153
- Tests 13
- The Annotated C++ Reference Manual (ARM) 167
- this 53
- thread 155
- Threads 152
 - Data Races 152
 - Ereignisse 156
 - Gültigkeitsbereiche 156
- throw 30
- to_string() 132
- to_ullong() 132
- try 30
- tuple 129, 133
- Typalias
 - Iterator 118
 - value_type 139
- Typen 5
 - abstrakte 41
 - complex 37
 - function 136
 - fundamentale 5
 - Größe 6
 - integrierte 17
 - Iteratoren 119
 - komplexe Zahlen 145
 - konkrete 36, 41
 - numerische Grenzen 148
 - parametrisierte 63
 - polymorphe 42
 - reguläre 67
 - sizeof 6
- Typfunktionen 137
- Typprädikate 140
- Typumwandlungen 172
 - auto 8

U

- Übereinstimmung
 - faule 88
 - gierige 88
- Überladen
 - Funktionen 5, 164
 - Operatoren 38, 164
- Überschreiben 43
 - explizites 47
- uniform_int_distribution 146
- Unions 21
 - diskriminierte 22
 - tagged 22
- unique_ptr 126
- unordered_map 111
- Unveränderlichkeit 9
- unwind (Aufrufstack abwickeln) 30
- using 28
- Utilities 125
 - Zeit- 171

V

- valarray 129
- <valarray> 148
- value_type 139
- Variablen 5
- variadische Templates 70
- Vec 107
- vector 103, 129
 - at() 107
 - Bereichsüberprüfung 106
 - Iteratoren 119
 - push_back() 104
- Vektoren
 - Arithmetik 148
 - Größen 104
 - valarray 148
- Veränderbarkeit 80
- Vererbung 43
 - Implementierungs- 48
 - Schnittstellen- 48
- Verkettung 79
- Verschiebekonstruktoren 54
- Verschieben
 - Algorithmen 123
 - Container 53

- Objekte 51
- Vector-Objekte 106
- Verschiebezuweisungen 55
- Verteilungen 146
 - exponential_distribution 146
 - Glockenkurve 146
 - normal_distribution 146
 - uniform_int_distribution 146
- virtual 42
- virtuelle Funktionen 42, 44
- virtuelle Funktionstabellen 44
- void* 177
- Vorbedingungen 30
- Vorlagen *siehe* Templates 63
- Vorwärts-Iteratoren 138
- vtbl 44

W

- Warteschlangen 157
- Wertargumente 65
- Werte 5
- Werttypen 139
- Wettlaufsituationen 157
- WG21 167
- while 13
 - Gültigkeitsbereich 40
- Wiederholungen 84
- Wildcards 84
- Wörterbuch 110

Z

- Zahlen
 - komplexe 145
 - numerische Grenzen 148
 - Zufalls- 146
- Zeichen
 - Newline 3
 - Sonderzeichen 3
- Zeichenklassen 85
 - Abkürzungen 86
- Zeichenkonstanten 93
- Zeiger 10
 - Elemente 103
 - gemeinsame 58
 - intelligente 58, 126
 - NULL 13
 - nullptr 12
 - shared_ptr 126
 - unique_ptr 126
- Zeit 134
- Zeitgeber 134
- Zufallszahlen 146
- Zusammenführen 123
- Zusicherungen *siehe auch* Assertionen 25, 32
- Zustandsmaschinen 83
- Zuweisungen
 - Kopier- 52
 - Verschiebe- 55