O'REILLY®

3. Auflage
Aktuell zu C++17

# C++

## kurz & gut

O'REILLYS TASCHENBIBLIOTHEK

Kyle Loudon
& Rainer Grimm

3. AUFLAGE

# C++
## *kurz & gut*

*Kyle Loudon & Rainer Grimm*

O'REILLY®

Kyle Loudon und Rainer Grimm

Lektorat: Alexandra Follenius
Korrektorat: Sibylle Feldmann, www.richtiger-text.de
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Ellie Volckhausen, Michael Oréal, www.oreal.de
Satz: III-Satz, www.drei-satz.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-078-6
PDF 978-3-96010-198-7
ePub 978-3-96010-199-4
mobi 978-3-96010-200-7

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

3. Auflage

Wieblinger Weg 17
69123 Heidelberg



5 4 3 2 1 0

# Inhalt

# Einführung

*C++ – kurz & gut* ist eine Schnellreferenz zum aktuellen C++-Standard C++17. Der internationale Standard ISO/IEC 14882:2017 umfasst gut 1.600 Seiten und wurde 2017 veröffentlicht. C++17 setzt die Tradition der C++-Standards fort, die mit C++11 und C++14 begonnen haben und im Dreijahreszyklus veröffentlicht werden. Der C++11-Standard, der 13 Jahre nach dem bis zu diesem Zeitpunkt einzigen C++-Standard C++98 verabschiedet wurde, steht für modernes C++. Formal betrachtet, ist C++03 zwar ein weiterer C++-Standard, der aber nur den Charakter einer technischen Korrektur besitzt.

Das Ziel dieser Kurzreferenz ist es, die Kernsprache von C++ kompakt vorzustellen. Trotzdem werden in diesem Buch aus reinem Pragmatismus einige Features aus der Standardbibliothek verwendet. Denn um ehrlich zu sein, ist die Sprache C++ ohne ihre Standardbibliothek nicht einmal eine halbe Sprache. Da die Komponenten der Standardbibliothek durch den Namensraum `std::` gekennzeichnet sind, sollte dies nicht zu Verwirrung führen. Zur C++-Standardbibliothek gehören die *Standard Template Library* (STL) mit den Klassen `std::string`, `std::vector` und `std::map`, die I/O-Streams mit den Objekten `std::cout`, `std::cerr` und `std::cin`, eine mächtige Bibliothek zum automatischen Speichermanagement, die neue Multithreading-Bibliothek und eine Bibliothek für reguläre Ausdrücke in C++, um nur die prominentesten zu nennen.

Dieses Buch wurde für Leser geschrieben, die bereits eine gewisse Vertrautheit mit C++ besitzen. Ein erfahrener C++-Programmierer wird aus der konzentrierten Referenz der Sprachmerkmale von C++ den größten Nutzen ziehen. Wenn Sie hingegen C++-Einstei-

ger sind, sollten Sie im ersten Schritt ein Lehrbuch dieser Kurzreferenz vorziehen. Haben Sie das Lehrbuch aber gemeistert, hilft Ihnen dieses Werk mit seinen vielen kurzen Codebeispielen, die Sprachmerkmale von C++ in einem weiteren Schritt sicher anzuwenden. Dadurch erwerben Sie solide C++-Kenntnisse und können sich anschließend in die Untiefen dieser Sprache vorwagen.

## Typografische Konventionen

In diesem Buch werden die folgenden typografischen Konventionen angewandt:

*Kursiv*
: Diese Schrift wird für Dateinamen und Hervorhebungen verwendet.

`Nichtproportionalschrift`
: Diese Schrift wird für Code, Befehle, Schlüsselwörter und Namen von Typen, Variablen, Funktionen und Klassen verwendet.

**Hinweis**

Dieser Abschnitt enthält weiterführende Informationen.

**Warnung**

Dieser Abschnitt warnt oder mahnt zur Vorsicht.

## Danksagungen zur 2. Auflage

Ich möchte Alexandra Follenius, meiner Lektorin bei O'Reilly, für ihre Unterstützung und Anleitung bei der Arbeit mit diesem Buch danken. Besonderer Dank gilt natürlich auch dem Autor der 1. Auflage dieses Werks, Kyle Loudon. Es war ein sehr spannendes Unternehmen, das Werk meines Vorgängers zu überarbeiten, um

all die neuen Features von C++11 einzuarbeiten. Ich danke vor allem aber Karsten Ahnert, Guntram Berti, Dmitry Ganyushin, Peter Gottschling, Sven Johannsen, Torsten Robitzki, Jörn Seger und Detlef Wilkening, die sich die Zeit genommen haben, das Manuskript auf sprachliche und insbesondere inhaltliche Fehler zu durchleuchten.

## 3. Auflage

Es freut mich sehr, die mittlerweile dritte Auflage dieses Buchs zu schreiben. Die dritte Auflage wird neben einigen Verbesserungen vor allem die zwei neuen Standards C++14 und C++17 enthalten. Features, deren Verhalten vom verwendeten C++-Standard abhängen oder die neu mit C++14 oder C++17 zum C++-Standard hinzugekommen sind, werde ich optisch hervorheben.

Dass die 3. Auflage dieses Buchs nur vier Jahre nach der 2. Auflage erscheint, zeigt vor allem eines: Die nun 40 Jahre alte Programmiersprache C++ ist immer noch dynamisch und erfreut sich großer Beliebtheit.

KAPITEL 1

# Programmstruktur

Auf der höchsten Ebene besteht ein C++-Programm aus einer oder mehreren *Quelldateien*, die C++-Quellcode enthalten. C++-Quelldateien binden oft zusätzlichen Quellcode in Form von *Header-Dateien* ein. Der C++-Präprozessor ist für das Einbinden des Codes aus diesen Dateien vor der Kompilierung jeder Datei zuständig. Gleichzeitig kann der Präprozessor aber auch mittels sogenannter *Präprozessoranweisungen* verschiedene andere Operationen ausführen. Eine Quelldatei wird nach der Vorverarbeitung durch den Präprozessor *Übersetzungseinheit* (Translation Unit) genannt.

## Programmstart

Die Funktion `main`, die genau einmal definiert werden muss, ist der Einstiegspunkt für das C++-Programm. In der standardisierten Form erwartet diese Funktion kein Argument oder aber zwei Argumente, die das Betriebssystem beim Programmstart mitliefert. Viele C++-Implementierungen erlauben zusätzliche Parameter. Der Rückgabetyp der `main`-Funktion ist `int`:

```
int main()
int main(int argc, char* argv[])
```

`argc` gibt die Anzahl der Argumente auf der Kommandozeile an. `argv` hingegen ist ein Array von C-Strings, das die Argumente in der Reihenfolge ihres Auftretens enthält. Der Name des ausführbaren Programms steht in dem Element `argv[0]`. Der Wert von `argv[argc]` ist 0.

Das folgende Beispiel zeigt die main-Funktion eines einfachen C++-Programms, das den Benutzer nach den Aktionen fragt, die auf einem Konto durchgeführt werden sollen:

```
#include <iostream>
#include <cstdlib>

#include "Account.h"

int main(int argc, char* argv[]){
  Account account(0.0);

  if (argc > 1) account.deposit(std::atof(argv[1]));

  char action;
  double amount;
  while (true){
    std::cout << "The balance is
              "<< account.getBalance()
              << std::endl;

    std::cout << "Choice: d, w or e: ";
    std::cin  >> action;

    switch (action){
      case 'd':
        std::cout << "Amount paid in: ";
        std::cin  >> amount;
        account.deposit(amount);
        break;
      case 'w':
        std::cout << "Amount paid out: ";
        std::cin  >> amount;
        account.withdraw(amount);
        break;
      case 'e':
        exit(0);
      default:
        std::cout << "Error" << std::endl;
    }
  }
  return 0;
}
```

Die Definition für die Klasse `Account` folgt später. Auf der Kommandozeile wird beim Programmstart eine initiale Einzahlung angegeben. Die Funktion `atof` aus der C++-Standardbibliothek dient dazu, das Kommandozeilenargument von einem C-String in einen `double`-Wert zu konvertieren.

## Programmende

Mit dem Verlassen der `main`-Funktion ist das C++-Programm beendet. Der Rückgabewert wird an das Betriebssystem weitergegeben und wird zum Rückgabewert des ausgeführten Programms. Falls `main` keine `return`-Anweisung enthält, wird ein implizites `return 0` am Ende des Funktionskörpers von `main` ausgeführt. Durch das explizite Aufrufen der `exit`-Funktion aus der C++-Standardbibliothek kann ein Programm direkt beendet werden. Diese Funktion erwartet den Rückgabewert des ausführbaren Programms als Argument.

## Header-Dateien

Header-Dateien, die typischerweise die Dateinamenserweitung *.h* besitzen, enthalten Quellcode, der in mehreren Dateien eingebunden werden kann. Eine Header-Datei sollte dagegen nie eines der folgenden Dinge enthalten:

- Definitionen von Variablen und statischen Attributen (der Abschnitt »Deklarationen« auf Seite 65 beschreibt den Unterschied zwischen Deklarationen und Definitionen).
- Definitionen von Funktionen mit Ausnahme von Funktions-Templates und Inline-Funktionen.
- Unbenannte Namensräume.

Header-Dateien in der C++-Standardbibliothek verwenden keine Dateinamenserweiterung. So ist die `std::string` in der Header-Datei *string* definiert.

Meistens wird für jede wichtige Klasse, die definiert wird, eine Header-Datei angelegt. Zum Beispiel ist die unmittelbar folgende Klasse `Account` in der Header-Datei *Account.h* definiert.

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account{
public:
  Account(double b);

  void deposit(double amt);
  void withdraw(double amt);
  double getBalance() const;

private:
  double balance;
};

#endif
```

Die Implementierung dieser Klasse befindet sich in der Datei *Account.cpp*. Durch die Präprozessoranweisung `#include` wird die Header-Datei in die Quelldatei eingebunden.

Da Header-Dateien wiederum häufig von anderen Header-Dateien eingebunden werden, ist es notwendig, darauf zu achten, dass ein und dieselbe Header-Datei nicht mehr als einmal eingebunden wird. Ansonsten kann dies zu Compiler-Fehlern führen. Durch das Verwenden der Präprozessoranweisungen `#ifndef`, `#define` und `#endif` in der Definition der Klasse `Account` wird das mehrfache Einbinden verhindert.

Die Strategie, die Header-Datei mit `#define` und `#endif` zu umgeben, zwingt den Prozessor, den Bezeichner mittels `#ifndef` abzufragen. Falls dieser nicht definiert ist, definiert der Präprozessor ihn und übersetzt den Inhalt der Datei. So wird in der Datei *Account.cpp* der Inhalt von *Account.h* nur dann übersetzt, wenn `ACCOUNT_H` nicht definiert ist. Das Erste, was während dieser Übersetzung passiert, ist die Definition von `ACCOUNT_H`, um sicherzustellen, dass die Header-Datei nicht noch ein zweites Mal übersetzt wird. Zur Gewährleistung der Eindeutigkeit wird per Konvention der Bezeichner *HEADER_H* (*Include-Guard*) verwendet. Dabei

steht *HEADER* für den Namen der Header-Datei ohne Dateinamenserweiterung.

Häufig werden *Include-Guards* verwendet, die mit einem Unterstrich (_) beginnen. Diese Namen sind jedoch für die Sprachimplementierung reserviert.

## Quelldateien

C++-Quelldateien enthalten C++-Quellcode. Sie haben in der Regel die Dateinamenserweiterung *.cpp*. Während der Kompilierung übersetzt der Compiler Quelldateien in Objektdateien, die typischerweise die Erweiterung *.obj* oder *.o* besitzen. Die Objektdateien werden vom Linker zusammengebunden, um ein fertiges, ausführbares Programm oder eine Bibliothek zu erzeugen.

Meist wird eine Quelldatei für jede wichtige Klasse anlegt. Beispielsweise findet sich die Implementierung von `Account` in der folgenden Datei *Account.cpp*:

```
#include "Account.h"

Account::Account(double b): balance(b){}

void Account::deposit(double amt){
  balance += amt;
}

void Account::withdraw(double amt){
  balance -= amt;
}

double Account::getBalance() const{
  return balance;
}
```

## Präprozessoranweisungen

Der C++-Präprozessor kann eine Reihe nützlicher Operationen durchführen, die über Anweisungen gesteuert werden. Jede Anwei-

sung beginnt mit einem Doppelkreuz (#) als erstem nicht leeren Zeichen einer Zeile. Präprozessoranweisungen können sich über mehrere Zeilen erstrecken, wenn ein Backslash (\) am Ende aller Zeilen mit Ausnahme der letzten steht.

## #define

Die Anweisung `#define` ersetzt einen Bezeichner, wo immer er auftritt, durch den darauffolgenden Text:

```
#define BUFFER_SIZE 80

char buffer[BUFFER_SIZE];  // Wird zu char buffer[80];
```

Wenn kein Text nach dem Bezeichner angegeben wird, definiert der Präprozessor den Bezeichner als leer. Damit lässt sich der Bezeichner aber abfragen, ob er definiert ist. Dieses Feature kam bereits bei der Definition von `ACCOUNT_H` zum Einsatz.

In C++ ist die Verwendung von Aufzählungen sowie von Variablen und Instanzvariablen, die mit den Schlüsselwörtern `const` oder `static const` definiert sind, der Anweisung `#define` für die Definition von Konstanten eindeutig vorzuziehen.

Die Anweisung `#define` kann Argumente zur Makroersetzung von Text entgegennehmen:

```
#define MIN(a, b) (((a) < (b)) ? (a):(b))

int x= 5, y= 10, z;

z= MIN(x, y);    // Setzt z auf 5.
```

Um unerwartete Probleme mit der Operatorpräzedenz zu vermeiden, sollten die Operatoren vollständig von Klammern umgeben werden.

In C++ sollten statt Makros Templates oder Inline-Funktionen verwendet werden. Templates und Inline-Funktionen verhindern unerwartete Ergebnisse von Makros vollständig. So wird durch `MIN(x++,y) x`

zweimal inkrementiert, falls das erste Argument kleiner als das zweite ist. Die Makroersetzung verwendet in diesem konkreten Fall x++ und nicht etwa das Ergebnis von x++ als ersten Parameter.

## #undef

Die #undef-Anweisung hebt die Definition eines Symbols auf, sodass ein Test auf Definition false ergibt:

```
#undef LOGGING_ENABLED
```

## #ifdef, #ifndef, #else, #endif

#ifdef, #ifndef , #else und #endif werden zusammen verwendet. Die #ifdef-Anweisung veranlasst den Präprozessor je nachdem, ob ein Symbol definiert ist oder nicht, unterschiedlichen Code einzubinden:

```
#ifdef LOGGING_ENABLED
std::cout << "Logging enabled" << std::endl;
#else
std::cout << "Logging disabled" << std::endl;
#endif
```

Die Verwendung von #else ist optional. #ifndef funktioniert ähnlich, aber der Code nach der #ifndef-Anweisung wird nur dann eingebunden, wenn das Symbol nicht definiert ist.

## #if, #elif, #else, #endif

Die Anweisungen #if, #elif, #else und #endif werden wie die #ifdef-Anweisungen zusammen verwendet. Sie veranlassen den Präprozessor, Code abhängig davon einzubinden, ob ein Ausdruck zu wahr oder zu falsch ausgewertet wird:

```
#if (LOGGING_LEVEL == LOGGING_MIN && LOGGING_FLAG)
std::cout << "Minimal Logging" << std::endl;
#elif (LOGGING_LEVEL == LOGGING_MAX && LOGGING_FLAG)
std::cout << "Maximal Logging" << std::endl;
#elif LOGGING_FLAG
std::cout << "Standard Logging" << std::endl;
#endif
```

Die #elif-(else-if-)Anweisung wird genutzt, um eine Reihe von Tests miteinander zu verketten.

## #include

Die #include-Anweisung veranlasst den Präprozessor, eine andere Datei, normalerweise eine Header-Datei, einzubinden. Standard-Header-Dateien werden in spitze Klammern, benutzerdefinierte in doppelte Anführungsstriche eingeschlossen:

```
#include <iostream>
#include "Account.h"
```

Der Präprozessor durchsucht je nachdem, ob die Datei in spitze Klammern oder Anführungszeichen eingeschlossen wird, unterschiedliche Pfade. Welche Pfade es sind, hängt vom System ab.

## #error

Die #error-Anweisung veranlasst das Anhalten der Kompilierung und die Ausgabe eines Texts:

```
#ifdef LOGGING_ENABLED
#error Logging should not be enabled
#endif
```

## #line

Die #line-Anweisung veranlasst den Präprozessor, die intern vom Compiler im Makro __LINE__ gespeicherte Zeilennummer zu ändern:

```
#line 100
```

Optional kann hinter der Zeilennummer ein Dateiname in doppelten Anführungsstrichen angegeben werden. Dadurch wird der intern im Makro __FILE__ gespeicherte Name geändert:

```
#line 100 "NewName.cpp"
```

### #pragma

Manche Operationen, die der Präprozessor durchführen kann, sind implementierungsabhängig. Mit der Anweisung `#pragma` lassen sich diese Operationen steuern, indem zusätzlich die jeweils benötigten Parameter angeben werden:

```
#ifdef LOGGING_ENABLED
#pragma message("Logging enabled")
#endif
```

## Präprozessormakros

Der C++-Präprozessor definiert mehrere Makros, um während der Kompilierung Informationen in eine Quelldatei einzufügen. Jedes Makro beginnt und endet mit zwei Unterstrichen. Einzige Ausnahme bildet `__cplusplus`, das keine Unterstriche am Ende hat.

`__LINE__`
: Wird zur aktuellen Zeilennummer in der kompilierten Quelldatei expandiert.

`__FILE__`
: Wird zum Namen der gerade kompilierten Quelldatei expandiert.

`__DATE__`
: Wird zum Datum der Kompilierung expandiert.

`__TIME__`
: Wird zur Uhrzeit der Kompilierung expandiert.

`__STDC__`
: Ist dann definiert, wenn der Compiler dem ANSI-C-Standard vollständig folgt.

`__cplusplus`
: Ist definiert, wenn das kompilierte Programm ein C++-Programm ist. Wie der Compiler bestimmt, ob es sich bei einem

bestimmten Programm um ein C++-Programm handelt, ist Compiler-spezifisch. Möglicherweise muss ein Compiler-Schalter angegeben werden, oder der Compiler verwendet die Dateinamenserweiterung der Quelldatei.

KAPITEL 2

# Lexikalische Elemente

Auf der fundamentalsten Ebene besteht ein C++-Programm aus einzelnen lexikalischen Elementen, sogenannten *Token*. Token sind logisch zusammenhängende Einheiten, die durch den Lexer gebildet werden, indem er den Textstrom in einen Tokenstrom zerlegt. Token werden in der Regel durch Leerraum (Leerzeichen, Zeilenwechsel, Tabulatoren usw.) voneinander abgegrenzt, können aber auch gebildet werden, wenn der Start des nächsten Tokens erkannt wird. Dies ist in dem nächsten Beispiel schön zu sehen:

```
ival+3
```

Dieser Tokenstrom besteht aus drei Token: `ival`, `+` und `3`. Wenn kein Leerraum vorhanden ist, bildet der Compiler die Token, indem er von links nach rechts nach der längstmöglichen logischen Einheit sucht.

Die Token werden an den Parser übergeben. Dieser bestimmt, ob der Tokenstrom die korrekte Syntax besitzt. Zusammen bilden die Token komplexere semantische Konstrukte wie Deklarationen, Ausdrücke und Anweisungen, die sich auf den Ausführungsfluss auswirken.

## Kommentare

Kommentare sind Anmerkungen im Quellcode. Sie richten sich an Entwickler und werden vom Compiler vollständig ignoriert. Sie werden in Leerzeichen konvertiert.

Ein Kommentar ist ein beliebiger Textblock, der entweder in `/*` und `*/` eingeschlossen wird oder hinter zwei Schrägstrichen (`//`) auf einer einzelnen Zeile folgt. Kommentare der ersten Form können