Die C++ Keywords

https://en.cppreference.com/w/cpp/keyword

Inhaltsverzeichnis

auto

- Bedeutung: Automatische Typableitung.
- Beispiel:

```
auto x = 10; // x ist vom Typ int
```

break

- Bedeutung: Beendet eine Schleife oder einen Switch-Fall.
- Beispiel:

```
for (int i = 0; i < 10; ++i) {
    if (i == 5) break;
}
```

case

- Bedeutung: Definiert einen Fall in einer Switch-Anweisung.
- Beispiel:

```
switch (x) {
    case 1:
        // Code für Fall 1
        break;
    case 2:
        // Code für Fall 2
        break;
}
```

catch

- Bedeutung: Fängt eine Ausnahme ab.
- Beispiel:

```
try {
    // Code, der eine Ausnahme werfen könnte
} catch (const std::exception& e) {
    // Ausnahmebehandlung
}
```

class

- Bedeutung: Definiert eine Klasse.
- Beispiel:

```
class MyClass {
public:
    int myVar;
};
```

const

- Bedeutung: Definiert eine Konstante.
- Beispiel:

```
const int x = 10;
```

constexpr

- Bedeutung: Definiert eine Konstante, die zur Kompilierzeit berechnet wird.
- Beispiel:

```
constexpr int square(int x) {
    return x * x;
}
```

continue

- Bedeutung: Überspringt den Rest der aktuellen Schleifeniteration und geht zur nächsten.
- Beispiel:

```
for (int i = 0; i < 10; ++i) {
    if (i == 5) continue;
}
```

default

- Bedeutung: Definiert den Standardfall in einer Switch-Anweisung.
- Beispiel:

```
switch (x) {
    case 1:
        // Code für Fall 1
        break;
    default:
        // Code für den Standardfall
        break;
}
```

delete

- Bedeutung: Gibt dynamisch allozierten Speicher frei.
- Beispiel:

```
int* p = new int;
delete p;
```

do

- Bedeutung: Beginnt eine Do-While-Schleife.
- Beispiel:

```
do {
    // Code
} while (condition);
```

double

- Bedeutung: Definiert eine Gleitkommazahl mit doppelter Genauigkeit.
- Beispiel:

```
double x = 10.5;
```

else

- Bedeutung: Definiert den alternativen Zweig einer If-Anweisung.
- Beispiel:

```
if (condition) {
    // Code für den wahren Fall
} else {
    // Code für den falschen Fall
}
```

enum

- Bedeutung: Definiert eine Aufzählung.
- Beispiel:

```
enum Color { Red, Green, Blue };
```

explicit

- Bedeutung: Verhindert implizite Typumwandlungen.
- Beispiel:

```
class MyClass {
public:
    explicit MyClass(int x) { /* ... */ }
};
```

export

- Bedeutung: Exportiert eine Template-Definition (selten verwendet).
- Beispiel:

```
export template <typename T> class MyClass { /* ... */ };
```

extern

- Bedeutung: Deklariert eine externe Variable oder Funktion.
- Beispiel:

```
extern int x;
```

false

- Bedeutung: Definiert den booleschen Wert falsch.
- Beispiel:

```
bool x = false;
```

float

- Bedeutung: Definiert eine Gleitkommazahl mit einfacher Genauigkeit.
- Beispiel:

```
float x = 10.5f;
```

for

- Bedeutung: Beginnt eine For-Schleife.
- Beispiel:

```
for (int i = 0; i < 10; ++i) {
    // Code
}
```

friend

- Bedeutung: Deklariert eine Freundfunktion oder -klasse.
- Beispiel:

```
class MyClass {
    friend void myFunction();
};
```

goto

- Bedeutung: Springt zu einer markierten Stelle im Code.
- Beispiel:

```
goto label;
// ...
label:
```

if

- Bedeutung: Beginnt eine If-Anweisung.
- Beispiel:

```
if (condition) {
    // Code für den wahren Fall
}
```

inline

- Bedeutung: Empfiehlt dem Compiler, eine Funktion inline zu expandieren.
- Beispiel:

```
inline int square(int x) {
    return x * x;
}
```

int

- Bedeutung: Definiert eine Ganzzahl.
- Beispiel:

```
int x = 10;
```

long

- Bedeutung: Definiert eine lange Ganzzahl.
- Beispiel:

```
long x = 10L;
```

mutable

- Bedeutung: Erlaubt die Änderung eines Mitglieds einer konstanten Klasse.
- Beispiel:

```
class MyClass {
    mutable int x;
};
```

namespace

- Bedeutung: Definiert einen Namensraum.
- Beispiel:

```
namespace MyNamespace {
    int x = 10;
}
```

new

- Bedeutung: Alloziert dynamisch Speicher.
- Beispiel:

```
int* p = new int;
```

noexcept

- Bedeutung: Deklariert eine Funktion, die keine Ausnahmen wirft.
- Beispiel:

```
void myFunction() noexcept {
    // Code
}
```

nullptr

- Bedeutung: Definiert einen Nullzeiger.
- Beispiel:

```
int* p = nullptr;
```

operator

- Bedeutung: Definiert einen Operator.
- Beispiel:

```
class MyClass {
public:
    MyClass operator+(const MyClass& other) { /* ... */ }
};
```

private

- Bedeutung: Definiert private Mitglieder einer Klasse.
- Beispiel:

```
class MyClass {
private:
    int x;
};
```

protected

- Bedeutung: Definiert geschützte Mitglieder einer Klasse.
- Beispiel:

```
class MyClass {
protected:
    int x;
};
```

public

- Bedeutung: Definiert öffentliche Mitglieder einer Klasse.
- Beispiel:

```
class MyClass {
public:
    int x;
};
```

register

- Bedeutung: Empfiehlt dem Compiler, eine Variable in einem Register zu speichern (selten verwendet).
- Beispiel:

```
register int x = 10;
```

reinterpret_cast

- Bedeutung: Führt eine Typumwandlung durch, die die Bitmuster neu interpretiert.
- Beispiel:

```
int* p = reinterpret_cast<int*>(0x1234);
```

return

- Bedeutung: Gibt einen Wert aus einer Funktion zurück.
- Beispiel:

```
int myFunction() {
    return 10;
}
```

short

- Bedeutung: Definiert eine kurze Ganzzahl.
- Beispiel:

```
short x = 10;
```

signed

- Bedeutung: Definiert eine vorzeichenbehaftete Ganzzahl.
- Beispiel:

```
signed int x = 10;
```

sizeof

- Bedeutung: Gibt die Größe eines Typs oder einer Variablen zurück.
- Beispiel:

```
int x = sizeof(int);
```

static

- Bedeutung: Definiert eine statische Variable oder Funktion.
- Beispiel:

```
static int x = 10;
```

static_assert

- Bedeutung: Führt eine Kompilierzeit-Assertion durch.
- Beispiel:

```
static_assert(sizeof(int) == 4, "int is not 4 bytes");
```

static_cast

- Bedeutung: Führt eine Typumwandlung durch, die zur Kompilierzeit überprüft wird.
- Beispiel:

```
double x = 10.5;
int y = static_cast<int>(x);
```

struct

- Bedeutung: Definiert eine Struktur.
- Beispiel:

```
struct MyStruct {
    int x;
};
```

switch

- Bedeutung: Beginnt eine Switch-Anweisung.
- Beispiel:

```
switch (x) {
    case 1:
        // Code für Fall 1
        break;
    case 2:
        // Code für Fall 2
        break;
}
```

template

- Bedeutung: Definiert eine Template-Funktion oder -Klasse.
- Beispiel:

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

this

- Bedeutung: Zeiger auf das aktuelle Objekt.
- Beispiel:

```
class MyClass {
public:
    void myFunction() {
        this->x = 10;
    }
private:
    int x;
};
```

thread_local

- Bedeutung: Definiert eine thread-lokale Variable.
- Beispiel:

```
thread_local int x = 10;
```

throw

- Bedeutung: Wirft eine Ausnahme.
- Beispiel:

```
if (condition) {
    throw std::runtime_error("Error occurred");
}
```

true

- Bedeutung: Definiert den booleschen Wert wahr.
- Beispiel:

```
bool x = true;
```

try

- Bedeutung: Beginnt einen Try-Block.
- Beispiel:

```
try {
    // Code, der eine Ausnahme werfen könnte
} catch (const std::exception& e) {
    // Ausnahmebehandlung
}
```

typedef

- Bedeutung: Definiert einen Typalias.
- Beispiel:

```
typedef unsigned long ulong;
```

typeid

- Bedeutung: Gibt den Typ einer Variablen zurück.
- Beispiel:

```
#include <typeinfo>
int x = 10;
const std::type_info& ti = typeid(x);
```

typename

- Bedeutung: Deklariert einen Typnamen in einer Template-Deklaration.
- Beispiel:

```
template <typename T>
class MyClass {
    typename T::iterator it;
};
```

union

- Bedeutung: Definiert eine Union.
- Beispiel:

```
union MyUnion {
    int x;
    float y;
};
```

unsigned

- Bedeutung: Definiert eine vorzeichenlose Ganzzahl.
- Beispiel:

```
unsigned int x = 10;
```

using

- Bedeutung: Deklariert eine Typalias oder importiert einen Namensraum.
- Beispiel:

```
using namespace std;
```

virtual

- Bedeutung: Deklariert eine virtuelle Funktion.
- Beispiel:

```
class Base {
public:
    virtual void myFunction() { /* ... */ }
};
```

void

- Bedeutung: Definiert einen leeren Typ.
- Beispiel:

```
void myFunction() {
    // Code
}
```

volatile

- Bedeutung: Deklariert eine volatile Variable.
- Beispiel:

```
volatile int x = 10;
```

wchar_t

- Bedeutung: Definiert eine Breitzeichen-Ganzzahl.
- Beispiel:

```
wchar_t x = L'a';
```

while

- Bedeutung: Beginnt eine While-Schleife.
- Beispiel:

```
while (condition) {
    // Code
}
```

alignas

- Bedeutung: Spezifiziert die Ausrichtung einer Variablen oder eines Typs.
- Beispiel:

```
alignas(16) int x;
```

alignof

- Bedeutung: Gibt die Ausrichtung eines Typs zurück.
- Beispiel:

```
int x = alignof(int);
```

and

- Bedeutung: Alternative Schreibweise für den logischen Und-Operator (&&).
- Beispiel:

```
if (a and b) { /* ... */ }
```

and_eq

- Bedeutung: Alternative Schreibweise für den bitweisen Und-Zuweisungsoperator (&=).
- Beispiel:

```
a and_eq b;
```

asm

- Bedeutung: Fügt Assemblercode ein.
- Beispiel:

```
asm("nop");
```

bitand

- Bedeutung: Alternative Schreibweise für den bitweisen Und-Operator (&).
- Beispiel:

```
int result = a bitand b;
```

bitor

- Bedeutung: Alternative Schreibweise für den bitweisen Oder-Operator (|).
- Beispiel:

```
int result = a bitor b;
```

bool

- Bedeutung: Definiert einen booleschen Typ.
- Beispiel:

```
bool x = true;
```

char

- Bedeutung: Definiert einen Zeichentyp.
- Beispiel:

```
char x = 'a';
```

char16_t

- Bedeutung: Definiert einen 16-Bit-Zeichentyp.
- Beispiel:

```
char16_t x = u'a';
```

char32_t

- Bedeutung: Definiert einen 32-Bit-Zeichentyp.
- Beispiel:

```
char32_t x = U'a';
```

compl

- Bedeutung: Alternative Schreibweise für den bitweisen Komplementoperator (~).
- Beispiel:

```
int result = compl a;
```

const_cast

- Bedeutung: Führt eine Typumwandlung durch, die die Konstanz oder Volatilität ändert.
- Beispiel:

```
const int x = 10;
int* p = const_cast<int*>(&x);
```

decltype

- Bedeutung: Gibt den Typ eines Ausdrucks zurück.
- Beispiel:

```
int x = 10;
decltype(x) y = 20;
```

not

- Bedeutung: Alternative Schreibweise für den logischen Nicht-Operator (!).
- Beispiel:

```
if (not condition) { /* ... */ }
```

not_eq

- Bedeutung: Alternative Schreibweise für den Ungleichheitsoperator (!=).
- Beispiel:

```
if (a not_eq b) { /* ... */ }
```

or

- Bedeutung: Alternative Schreibweise für den logischen Oder-Operator (||).
- Beispiel:

```
if (a or b) { /* ... */ }
```

or_eq

- Bedeutung: Alternative Schreibweise für den bitweisen Oder-Zuweisungsoperator (|=).
- Beispiel:

```
a or_eq b;
```

xor

- Bedeutung: Alternative Schreibweise für den bitweisen Xor-Operator (^).
- Beispiel:

```
int result = a xor b;
```

xor_eq

- Bedeutung: Alternative Schreibweise für den bitweisen Xor-Zuweisungsoperator (^=).
- Beispiel:

```
a xor_eq b;
```