



BOOKLET

# DIE NEUEN C++-STANDARDS – C++11 BIS C++20

## PROFITIEREN SIE VON UNSERER ERFAHRUNG!

### Kontakt Schweiz

bbv Software Services AG  
Blumenrain 10  
6002 Luzern  
Telefon: +41 41 429 01 11  
E-Mail: info@bbv.ch

### Kontakt Deutschland

bbv Software Services GmbH  
Agnes-Pockels-Bogen 1  
80992 München  
Telefon: +49 89 452 438 30  
E-Mail: info@bbv.eu

Der Inhalt dieses Booklets wurde mit Sorgfalt und nach bestem Gewissen erstellt. Eine Gewähr für die Aktualität, Vollständigkeit und Richtigkeit des Inhalts kann jedoch nicht übernommen werden. Eine Haftung (einschliesslich Fahrlässigkeit) für Schäden oder Folgeschäden, die sich aus der Anwendung des Inhalts dieses Booklets ergeben, wird nicht übernommen.

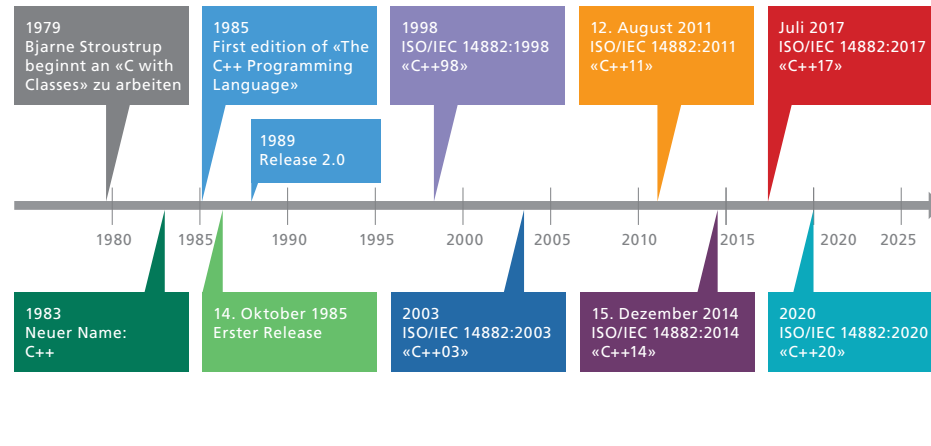
## INHALT

<b>1</b>	<b>Einleitung</b>	<b>5</b>
<b>2</b>	<b>Spracherweiterungen</b>	<b>7</b>
	Vereinheitlichte Initialisierung	8
	Typeninferenz	12
	Enums	14
	Keywords für Klassen & Klassenmember	17
	Explicit-Operator	23
	Static Assertions	24
	Noexcept Operator	25
	Dreiwegvergleich Operator	27
	Keyword nullptr	30
	Right Angle Brackets	31
	Range Based For Loops	32
	Nested Namespace Definitions	33
	Trailing Return Type	34
	Lambda-Funktionen	35
	Konstante Ausdrücke	39
	Alignment	48
	Move Semantics	51
	Coroutine	58
	Variadic Templates	60
	Concepts	64
	Return Type Deduction und decltype(auto)	68
	Structured Bindings	70
	Selection Statements mit Initializer	71

Standardattribute und Annotations	72
Einführung weiterer Syntaxerweiterungen	78
Lockerungen Kontext bezogener C++-Umwandlungen	80
Modules	81
Has_include Präprozessor-Direktive	85
Anpassung zur Codeoptimierung	87
<b>3 Standard-Library-Erweiterungen</b>	<b>88</b>
<b>4 Fazit</b>	<b>89</b>
<b>5 Anhang</b>	<b>91</b>
Autoren	92
Hinweis zur Kompilierfähigkeit der Beispiele	92
Quellenverzeichnis	93
Literatur	95

# 1 EINLEITUNG

C++ wurde von 1979 an von Bjarne Stroustrup als Erweiterung der Programmiersprache C mit dem Namen «C with Classes» entwickelt. Wie der Name schon sagt, war die erste und wichtigste Erweiterung das Klassenkonzept mit Datenkapselung. Im Jahre 1983 wurde dann der Name C++ eingeführt. Die erste Referenzversion von C++ erschien 1985, die aber noch nicht standardisiert war. Kurz darauf erschien 1989 die Version 2.0 von C++ mit den Neuerungen: Mehrfachvererbung, abstrakte Klassen, statische Elementfunktionen usw. Die endgültige Fassung der Sprache C++ nach ISO/IEC 14882:1998 wurde 1998 genormt. 2003 verabschiedete man eine Nachbesserung der Norm von 1998 als «C++03» nach ISO/IEC 14882:2003. Seit 2011 werden die C++ als Sprache sowie der ISO/IEC-14882-Standard alle drei Jahre erweitert. Nach der Einführung von C++17 konnte ein enormer Anstieg an Beiträgen zur Spracherweiterung verzeichnet werden, der schliesslich zu C++20 führte.



Die wichtigsten Elemente dieser Fassung werden im Kapitel «Spracherweiterungen» anhand einer kurzen Beschreibung und der Syntax mit entsprechenden Beispielen erläutert. Im folgenden Kapitel wird die Standard Library nur kurz erwähnt, da es den Rahmen des Booklets übersteigen würde, und dafür auf Online-Informationen verwiesen.

Dieses Booklet ist für Softwareentwickler gedacht, die sich mit den neuen C++-Standards auseinandersetzen wollen. Es gibt einen Überblick über die neuen, modernen Funktionen und Spracherweiterungen, die dem Softwareentwickler das Leben erleichtern können bzw. sollen.

## 2 SPRACHERWEITERUNGEN

## 2.1 VEREINHEITLICHTE INITIALISIERUNG C++11

Der neue Standard vereinheitlicht die Benutzung der geschweiften Klammern bei der Initialisierung, das heisst, sie ist frei von Mehrdeutigkeiten, und bei gleichem Aussehen bedeutet es auch das Gleiche. Alle bisherigen Initialisierungskonstrukte bleiben gültig.

**Syntax:** `expression { expression };` oder  
`expression = { expression };`

### Beispiel:

```
// simple initializing using {}
Klass k{123, "b"};
int x{42};

std::vector<int> v{1, 2, 3, 4}; // list initialisation
std::vector<int> y = {0, 1, 2, 3, 4};

int fail{7.5f}; // fails because of narrowing down from float to int
```

### Vereinheitlichte Initialisierung von Aggregates:

- Members werden in der Reihenfolge der Deklaration **copy initialized**
- Implizite Konvertierungen sind erlaubt
- Leere Initialisierungslisten bedeuten **value initialized**
- Initialisierungslisten können weniger Elemente haben als benötigt werden

### Beispiel:

```
// initializing of aggregates
struct Data {
    int a{0};
    double b{0};
    std::string c;
};
Data data{7, 3.14, "Seven Pies!"};
// Error narrowing from double to int
Data data2{7.0, 3.14, "Seven Pies!"};
```

### Initialisierung von Non-aggregates:

- Argumente von Initialisierungslisten werden dem Konstruktor übergeben
- Implizite Konvertierungen sind erlaubt, aber **narrowing-Konvertierungen** nicht

### Beispiel:

```
struct MyType {
    // the constructor makes a non-aggregate out of this struct
    MyType(const std::string &msg, int x) : m_a(x), m_c(msg) {}
    int m_a;
    double m_b;
    std::string m_c;
};
MyType myType{"Fortytwo!", 42};
```

**Initialisieren von Container-Typen:**

- Argumente von Initialisierungslisten müssen alle vom selben Typ sein, implizite Konvertierungen sind erlaubt, aber keine **narrowing**-Konvertierungen
- Funktioniert für Container mit statischer oder dynamischer Grösse

**Container-Typen** definieren einen Konstruktor mit

```
std::initializer_list<T>
```

**Beispiel:**

```
// Example for a container-type with explicit list
initialisation
struct Bar {
    explicit Bar(std::initializer_list<std::string> in) {
        for (auto &s : in)
            m_data.push_back("Bar " + s);
    }
    std::vector<std::string> m_data;
};
Bar bar{"A", "B", "C", "D"};
```

**Initialisierung von nicht statischen Data-Memberrn:**

- Nicht statische **Data Members** können jetzt zum Zeitpunkt der Deklaration initialisiert werden
- PODs werden dadurch in C++14 zu non PODs

**Beispiel:**

```
struct Klass {
public:
    Klass(int x, const std::string &s) : _x(x), _s(s) {}

private:
    // initialisation can be specified at the time of declaration

    int _x{999}; // Value assignment through construction
    int _y = 1000; // value assignment
    std::string _s;
};
```

## 2.2 TYPENINFERENZ

C++11

### 2.2.1 Keyword auto

Mit dem Keyword `auto` bestimmt der Compiler automatisch den optimalen Typ für den Ausdruck. Verwendung zur expliziten Initialisierung und bei Templates für bessere Lesbarkeit. Bemerkung: Das Keyword `auto` existierte bereits bei C++03, allerdings als storage-specifier wie `extern`, `register` oder `static` und ist somit nicht abwärtskompatibel.

**Syntax:** `auto expression;`

#### Beispiele:

```
auto x = 42; // x is an int
x = "fortytwo"; // Error, because x is already an int
std::vector<int> vec{1000, 1001, 1002, 1003};
auto it0 = vec.begin(); // it0 is a vector<T>::iterator
auto st0(vec[4]); // st0 is an int
auto &st1 = vec[4]; // st1 is an int&
const auto &s1 = vec[4]; // s1 const int&
auto it1 = vec.cbegin(); // it1 is a vector<int>::const_iterator
auto sil{vec[4]}; // sil is a std::initializer_list<int>
```

### 2.2.2 Decltype-Operator

Der `decltype`-Operator kann dazu verwendet werden, um Variablen abhängig von einem Ausdruck ohne explizite Initialisierung zu deklarieren. Zum Beispiel, wenn Funktionen mit einem unbekanntem Rückgabebetyp definiert sind.

**Syntax:** `decltype(expression) name;`

#### Beispiele:

```
const std::string &other{"Abracadabra"};
const std::string &name();

// using decltype to "copy" the type from an existing object
decltype(name()) n1(other); // n1 contains the type 'const std::string&'

std::string s1;
decltype(s1) s2; // s2 contains the type std::string
struct Foo {
    int i;
};
Foo foo;
decltype(foo.i) j; // j contains type int
decltype((foo.i)) k(j); // k contains type int&, because
// (foo.i) is an expression which has an address
```

## 2.3 ENUMS

C++11

Ein `enum` in Verbindung mit `class` ist komplett typensicher und seine Elemente werden nicht mehr automatisch von und nach `int` konvertiert. Ein weiterer Vorteil ist, dass spezifiziert werden kann, welcher `int`-Typ dem `enum` zugrunde liegt (default ist `int`). Da die Elemente eines `enums` nicht mehr im äusseren `namespace` angesiedelt sind, verursachen sie deshalb keine Namenskonflikte mehr und müssen über ihren Klassennamen angesprochen werden.

**Syntax:** `enum class expression;`

Die Elemente von «alten» `enums` können nun zusätzlich über ihren vollen Namen angesprochen werden.

### Beispiel:

```
// non scoped enum
enum Color { Cyan, Magenta, Yellow, Black };
enum Sound { Beep, Bop, Boing };

// the keyword 'class' following 'enum' declares a scoped enum
enum class RGB { Red, Green, Blue, Black };
enum class CMY { Cyan, Magenta, Yello, Black };

// scoped enum with explicit type specifier
enum class Fruits : unsigned int { Apples, Pears };

CMY cmy = CMY::Black; // CMY scope
RGB rgb = RGB::Black; // RGB scope

Color c1 = Cyan;      // ok
Color c2 = Color::Cyan; // also ok
RGB red = Red;      // Error unscoped assignment

// Error, cross-scope comparison
if (rgb == cmy) {
}

// Compiles with only a warning but is semantically incorrect
if (Sound::Boing == Color::Yellow) {
}
```



### 2.3.2. Using enum

C++20

Seit C++20 können **enums**, ähnlich wie Namespace, mit einer **using**-Direktive in einen Scope eingebunden werden. Die Elemente von «alten» **enums** müssen nun nicht mehr über ihren vollen Namen angesprochen werden. Dies erhöht die Lesbarkeit des Codes.

**Syntax:** `using enum <name>;`

**Beispiel:**

```
// the keyword 'class' following 'enum' declares a scoped
enum class RGB { Red, Green, Blue};

std::string GetColorName(RGB color) {
    switch (color) {
        using enum RGB; // OK with C++20
        case Red:     return "Red";
        case Green:   return "Green";
        case Blue:    return "Blue";
    }
    return "";
}
```

### 2.4 KEYWORDS FÜR KLASSEN & KLASSENMEMBER

C++11

#### 2.4.1 default

Eine sehr nützliche Erweiterung von C++11 ist **default**. Damit kann der Compiler angewiesen werden, eine **default**-Implementation eines Konstruktors oder Zuweisungsoperators sowie von trivialen Destruktoren zu generieren.

**Beispiel:**

```
struct Dummy {
    Dummy() = default;
    // specify destructor as virtual but use default implementation
    virtual ~Dummy() = default;

    Dummy(int x) = default; // error this is not a default constructor

protected:
    // protect move, but use default implementation
    Dummy(Dummy &&other) = default;
    Dummy &operator=(Dummy &&other) = default;

    // protect copy, but use default implementation
    Dummy(const Dummy &other) = default;
    // also protect assignment operator
    Dummy &operator=(const Dummy &rhs) = default;
};
```

### 2.4.2 delete

Mit `delete` können Funktionen und Konstruktoren von Klassen entfernt werden. Dies vereinfacht die Definition von Klassen, denen der Compiler automatisch die `default`-Implementierungen hinzufügt.

#### Beispiel:

```
struct DummyNonCopyable {
    DummyNonCopyable() = default;
    // disables copying the object through construction
    DummyNonCopyable(const Dummy &) = delete;
    // disables copying the object through assignment
    DummyNonCopyable &operator=(const Dummy &rhs) = delete;
};

struct NonDefaultConstructible {
    // this struct can only be constructed through a move or copy
    NonDefaultConstructible() = delete;
};

struct NonContstructableByValue {
    void f(int){};
    void f(double) = delete; // prevents f.foo(3.14)
};
```

### 2.4.3 final

Mit `final` werden Methoden und Klassen vor weiteren Ableitungen geschützt.

#### Beispiel:

```
struct NonDeriveable final {};
```

// Error cannot derive from Base because of final

```
struct Derived : NonDeriveable {};
```

```
struct BaseWithFinalMembers {
    virtual void foo(int) final;
    virtual void bar();
    void nonVirtual();
};

struct Derived2 : public BaseWithFinalMembers {

    void foo(int) override; // error because of foo() being 'final' in base class
}
```

### 2.4.4 override

Mit **override** muss eine Methode der Basisklasse überschrieben werden. Es schützt davor, eine Methode zu definieren, die aus Versehen nichts überschreibt. Dabei muss die ganze Methodensignatur übereinstimmen. Fehler werden nun bereits durch den Compiler angezeigt.

#### Beispiel:

```
struct Derived3 : public BaseWithFinalMembers {}

// error does not override any function of base class
void nonexistent() override;

// error nonVirtual is not marked virtual in the base class
void nonVirtual() override;
void bar() override; // explicitly override bar

};
```

### 2.4.5 Konstruktorenweiterleitung

C++17

Seit C++17 können Konstruktoren definiert werden, die andere Konstruktoren aufrufen – die Initialisierung also delegieren. Durch das Delegieren an andere Konstruktoren können viele Codeduplikationen vermieden werden.

#### Beispiel:

```
class DelegatingCtor {
    int m_number;

public:
    DelegatingCtor(int new_number) : m_number(new_number) {}
    DelegatingCtor()
        : DelegatingCtor(42) {} // constructor delegates to
    // DelegatingCtor(int)
};
```

### 2.4.6 Inheriting constructors

Normalerweise erbt eine Klasse die Konstruktoren von ihrem Vorgänger nicht. Mit `using` können die Konstruktoren des Vorgängers zu den eigenen hinzugefügt werden.

#### Beispiel:

```
struct Base {
    Base() = default;
    // explicit to avoid implicit conversion from ptr-types
    explicit Base(int z){};
};
struct InheritingCtor : public Base {
    using Base::Base;           // Inherit all ctors from Base
    InheritingCtor(int x, int y){}; // Additional Ctor
};
InheritingCtor d1(5, 10); // Uses additional ctor
InheritingCtor d2(42);   // Uses inherited ctor
```

### 2.5 EXPLICIT-OPERATOR

C++11

Implizite Konvertierungen sind nicht immer erwünscht, deshalb lässt sich die implizite Konvertierung mit `explicit` unterbinden. Mit C++98 war dies nur für Konstruktoren möglich. Neu kann mit C++11 auch der Konvertierungsoperator `explicit` deklariert werden.

**Syntax:** `explicit expression`

#### Beispiel:

```
struct ExplicitConversion {
    // Conversion to bool overwritten
    explicit operator bool() { return true; }
};

ExplicitConversion myBool;
if (myBool) {
} // OK
int a = (myBool) ? 3 : 4; // OK
int b = myBool + a; // Error, only conversion allowed
```

## 2.6 STATIC ASSERTIONS

C++11

`static_assert` überprüft konstante Ausdrücke zur Übersetzungszeit. Wenn die Auswertung des Ausdrucks nicht `true` ergibt, bricht der Compiler mit der angegebenen Fehlermeldung ab. Static assertions sind bei der Programmierung von Templates sehr hilfreich.

**Syntax:** `static_assert(bool const expression, error message);`

**Beispiele:**

```
static_assert(sizeof(long) == 8, "64 bit code not supported!");

template <typename T> struct Dummy {
    static_assert(sizeof(int) >= sizeof(T), "Dummy<T>: T is too small!");
};

Dummy<int> di; // OK

Dummy<short> ds; // Assert on compile time
```

## 2.7 NOEXCEPT OPERATOR

C++11

Das keyword `noexcept` ist einerseits als Specifier für Funktionen benutzbar, um anzuzeigen, ob eine Funktion `exceptions` werfen kann oder nicht, andererseits als Operator, um dieses Verhalten zur Übersetzungszeit zu überprüfen.

Der Specifier überprüft zur Übersetzungszeit nicht, ob eine markierte Funktion tatsächlich keine `throw`-Direktive enthält. Fehlt der Specifier wird immer `noexcept(false)` angenommen. Standard-Konstrukturen und Destruktoren sind immer `noexcept(true)`.

Der `throw()`-Specifier wurde mit C++11 als überholt markiert, wird aber in den meisten Compilern noch für `noexcept`-Abfragen unterstützt.

Dynamische Typenspezifikation mit `throw(type)` ist seit C++17 nicht mehr erlaubt und führt zu einem Fehlschlag der Kompilierung. Der Operator `noexcept(funktion)` führt die Funktion nicht aus, sondern überprüft nur die Spezifikation des `exceptions`-Verhaltens.

**Beispiel:**

```
class A {
public:
    void throwing_function() noexcept(false) {
        // marked as possibly throwing
        if (_a < 500) {
            throw myException();
        }
    }
    void exceptionless_func() noexcept {}
    void bad_func() noexcept {
        // putting a throw-statement here will NOT break
    }
};
```

```

        compilation even if it violates the noexcept specifier
        throw myException();
    }
    // the same as noexcept(false)
    void unspecified_func() {}

    // deprecated since C++11 but the same as noexcept
    void old_syntax_func() throw() {}

    // deprecated since C++11 but the same as noexcept(false)
    // dynamic exception specifications are no longer allowed in C++17
    // void old_syntax_throwing_func() throw(myException) { throw
        myException(); }

private:
    int _a{1000};};int main(int argc, char **argv) {
    A a;
    static_assert(noexcept(a.exceptionless_func()), "not throwing");
    static_assert(!noexcept(a.throwing_function()), "not throwing");
    static_assert(!noexcept(a.unspecified_func()), "possibly throwing");
    static_assert(noexcept(a.bad_func()), "bad func pretends to be not
        throwing");
    static_assert(noexcept(a.old_syntax_func()), "not throwing");
    static_assert(!noexcept(a.old_syntax_throwing_func()), "possibly
        throwing");
}

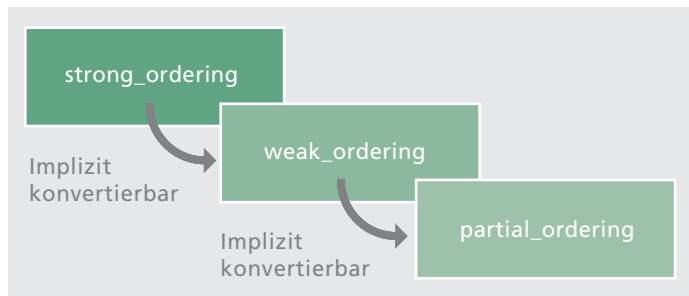
```

## 2.8. DREIWEGERGLEICH OPERATOR <=> C++20

Der Operator <=>, auch bekannt als «Spaceship Operator» (oder TIE-Interceptor), dient zum einfacheren Vergleich von Objekten und ersetzt formal die Operatoren ==, !=, <, <=, >, >=. Der Operator unterstützt damit auch das Sortieren von Objekten. Vor allem die Notwendigkeit des Überladens einzelner Operatoren entfällt dadurch, weil sie durch den Operator <=> abgebildet werden können. Probleme wie Overflow und Konflikte mit DRY werden vermieden.

Dreiwegevergleich bedeutet, dass der Operator <=> alle drei Kombinationen <, ==, und > abbildet (vgl. `strcmp`). Das Resultat hängt davon ab, ob die zu vergleichenden Typen auch sortierbar sind, und welchen Wertebereich sie einnehmen können.

Beim Vergleich von integralen Typen wird das Resultat als `strong_ordering` bezeichnet. Damit wird ausgedrückt, dass die Werte in jedem Fall vergleichbar und sortierbar sind. Eine `weak_ordering` liegt vor, wenn der Zustand der Gleichheit nicht eindeutig ist (Gesetz der Substituierbarkeit), zum Beispiel Rechtecke gleichen Flächeninhalts, aber mit unterschiedlich langen Seiten. Eine `strong_ordering` kann implizit in eine `weak_ordering` konvertiert werden. Eine `partial_ordering` ist noch schwächer und deckt auch die Fälle NaN für float ab. Ursprünglich waren im Entwurf des C++20 Standards noch die `strong_equality` und `weak_equality` vorgesehen, wurden aber später (mit P1959R0) wieder entfernt.



Alle Typen haben gemeinsam, dass beim Vergleich von A und B gilt:

- $A <=> B > 0$ , wenn  $A > B$
- $A <=> B == 0$ , wenn  $A == B$
- $A <=> B < 0$ . Wenn  $A < B$

#### Beispiel:

```

#include <compare>

// The float radius requires a partial_ordering

struct Circle {
    float radius;
    std::partial_ordering operator<=>(const Circle &rhs) {
        return radius <=> rhs.radius;
    }
};

Circle circle_a(1.42f);
Circle circle_b(1.23f);
std::cout << std::boolalpha;
std::cout << "a <=> b > 0: " << (circle_a <=> circle_b > 0)
    << std::endl;
// Output: "a <=> b > 0: true"
  
```

Es sei noch explizit darauf hingewiesen, dass exklusiv der Vergleich mit dem Literal '0' zugelassen ist. Jeder andere Vergleich ist per Definition **undefined behaviour**.

Ein wesentliches Feature des Operators ist die Deklaration mit dem Keyword **=default**. Dadurch werden alle anderen sechs Vergleichsoperatoren automatisch vom Compiler generiert. Der Code wird um einiges übersichtlicher. Die Operator  $a==b$  entspricht dann beispielsweise der Operation  $a<=>b == 0$ .

#### Beispiel:

```

struct SmallCompare {
    // instead of repeating much code and define all
    // operators ==, <=, ... just do this:
    auto operator<=>(const ClassWithManyMembers &rhs) const
        = default;
    // can now be compared by ==, !=, <=, >=, <, >
};
  
```

Mit C++20 werden ausserdem die **concepts three\_way\_comparable** und **three\_way\_comparable\_with** eingeführt, die für die Deklaration von generischen Klassen genutzt werden kann (siehe Concepts) bzw. mit dem struct **compare\_three\_way** bereits genutzt wird.

Für den Fall, dass eine Klasse den Operator **<=>** nicht unterstützt, wurde ausserdem in **<algorithm>** noch die Funktion **compare\_3way** definiert. Diese Funktion beschränkt sich dann auf die Operatoren **<** und **==**.

## 2.9 KEYWORD NULLPTR

C++11

Das Schlüsselwort `nullptr` ist die typensichere Variante von `NULL`. Im Normalfall ist die Verwendung identisch mit der von `NULL`. `nullptr` konvertiert implizit zu jedem Pointer, Pointer-to-Member-Typ sowie dem `bool`-Typ.

### Beispiel:

```
// this is how NULL used to be defined. NULL was never a language keyword, but
// always a predefined macro
#define NULL 0

void foo(const char *dummy) {

    if (dummy == nullptr) {
        // do something
    }
}

void foo(int i) {}

void bar(const char *c) {}
void bar(const float *f) {}

int main(int, char **) {

    const char *dummy = nullptr; // OK
    // error, except if direct initialisation enabled during compilation
    bool valid = nullptr;
    int value = nullptr; // Error, assigning a pointer type to int
    int anInt{42};
    int *ptrTo_anInt = &anInt;
    ptrTo_anInt = nullptr; // OK

    foo(NULL); // calls foo(int)
```

```
foo(nullptr); // calls foo(const char*), because it matches the ptr type

bar(nullptr); // ambiguous call to bar(const int*) or bar(const float*)
bar(static_cast<const float *>(nullptr)); // force pointer type
}
```

## 2.10 RIGHT ANGLE BRACKETS

C++11

Bis C++03 wurde `>>` in allen Fällen als «right shift»-Operator interpretiert. C++11 verbessert die Spezifikation der Parser, sodass mehrfache **right angle brackets** als Abschluss der Template-Argumente interpretiert werden, wo es begründet ist.

### Beispiele:

```
template<bool T>
class SomeType
{

};

//OK before and after C++11. Note the space between the >>
std::vector<std::pair<int, int> > vec0;

//OK only after C++11
std::vector<std::pair<int, int>> vec1;

//OK after C++11: interpreted as std::vector<SomeType<false>>
std::vector<SomeType<(1>2)>> vec3;
```



## 2.11 RANGE BASED FOR LOOPS C++11

Die neue Syntax erlaubt das Schreiben von `for` loops über alle seine Elemente in kürzerer Form. Die Syntax steht für folgende Sprach-elemente zur Verfügung:

- Arrays bekannter Grösse
- Initialisierungslisten mit vereinheitlichter Klammersyntax
- Standard-Container inklusive `string` mit deren Methoden `begin()` und `end()`
- eigene Klassen mit Methoden `begin()` und `end()`
- Überladungen der freien Funktionen `begin()` und `end()`

**Syntax:** `for (expression) { ... }`

**Beispiel:**

```
int iArray[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

// access the elements by value
for (int i : iArray) {
    // do something
};

std::vector<int> v{1, 2, 3, 4};

// access the elements as references
for (const auto &i : v) {
    // do something
}
```

## 2.12 NESTED NAMESPACE DEFINITIONS C++17

Seit C++17 können verschachtelte Namespaces in einer kürzeren Variante definiert werden.

**Beispiel:**

```
namespace A::B::C {
void doMagic() {
}
}

int main(int argc, char **argv) {
    A::B::C::doMagic();

    return 0;
}
```

## 2.13 TRAILING RETURN TYPES

C++11

Trailing return types sind eine alternative Syntax für Funktionsdeklarationen. Sie sind ein sehr leistungsfähiges Instrument bei Template-Funktionsdeklarationen. Der Rückgabebetyp wird anhand eines Ausdrucks abhängig von den Typen der Argumente einer Funktion ermittelt (siehe dazu auch das Folgekapitel **Lambda-Funktionen**).

**Syntax:** `auto function(T1 a, T2 b) -> T3`

**Beispiel:**

```
struct A {
    struct B {
        struct C {
            struct D {};

            D foo(const C &); // returns an instance of A::B::C::D
            D bar(const C &); // returns an instance of A::B::C::D
        };
    };
};

// Standard, the scope of D has to be specified
A::B::C::D
A::B::C::foo(const C &) { return D(); };

// Trailing return type, D is in scope of A::B::C
auto A::B::C::bar(const C &) -> D { return D(); };

// automatic computation of the return type
template <typename A, typename B> auto add(A a, B b) -> decltype(a + b) {
    return a + b;
}
```

## 2.14 LAMBDA-FUNKTIONEN

C++11

**Lambda-Funktionen** sind anonyme Funktionen und können dort definiert werden, wo sie gerade gebraucht werden.

**Syntax:** `[Capture Clause] (Parameter) -> Type {Function Body}`

`[]` umschließt die Zugriffsdeklaration; definiert wird, auf welche Variablen der Umgebung im **Lambda-Funktionskörper** zugegriffen werden kann. Seit C++17 kann innerhalb von Klassenmembern auch `this` als **capture** übergeben werden.

`(...)` ist die Deklaration der Argumente; hat die **Lambda-Funktion** keine Argumente, kann dies weggelassen werden. Seit C++14 sind auch generische Lambdas mit **'auto'** als Parameter-Typ zugelassen.

`->` ... ist optional und dient zur Angabe des Rückgabetyps; wird dieser weggelassen, ermittelt der Compiler den Typ automatisch.

`{ ... }` enthält den eigentlichen Funktionskörper; es ist alles erlaubt, was auch innerhalb von normalen Funktionen erlaubt ist.

**Beispiele:**

```
// declaration of a lambda function
const auto square = [](int x) { return x * x; };
// execution of the lambda above
const auto r = square(4);

// the parameter brackets () can be omitted if empty set
const auto abbreviated = [] { return 55; };

// using auto a parameter types for even more generic programming
const auto auto_param_function = [](const auto &a, const auto &b) {
    return a * b;
};
```

Verschiedene Zugriffsdeklarationen auf Variablen von **Lambda**-Funktionen:

```
int x, y, z;
const auto no_capture = []() {}; // No variables are captured
const auto capture_by_value = [x, y]() {}; // x and y are captured by value
const auto capture_by_ref = [&x, &y]() {
}; // x and y are captured by reference
const auto mixed_capture = [x, &y, &z]() {}; // mixed captures

// captures all variables in scope by value (square, r, x,y,z)
const auto capture_all_by_value = [=]() {};
// all variables in scope by reference
const auto capture_all_by_ref = [&]() {};

// all variables in scope by value, except x which is captured by
reference
const auto capture_all_by_value_except = [=, &x]() {};
```

```
// capture all by reference except x
const auto capture_all_by_ref_except = [&, x]() {};

// initialized lambda captures
auto init_function = [&r = x, x = x + 1]()->int {
    r += 2;    return x + 2;
};
```

```
// using lambda with std::-functions
std::vector<int> a1{1, 2, 3, 4, 5};
std::vector<int> a2(a1.size());
// fills a vector with the squares of the values of another vector
std::transform(a1.begin(), a1.end(), a2.begin(), square);

// same but with explicit trailing return type
std::transform(a1.begin(), a1.end(), a2.begin(),
    [](int x) -> int { return x * x; });
std::vector<int> v{1, 2, 3, 4, 5}; int sum{0};
// sums up the values of vector v using pass by reference for sum
std::for_each(v.begin(), v.end(), [&sum](int i) { sum += i; });
```

Zusammenfassung:

- Die anonymen **Lambda**-Funktionen vereinfachen die Programmierung von Standard-Library-Algorithmen und Funktoren
- In Fällen, wo der return-Typ spezifiziert werden muss, ist die **trailing-return-type**-Syntax anzuwenden
- Es gibt keine Beschränkungen für die Komplexität von **Lambda**-Funktionen. Sie sollten trotzdem einfach gehalten werden, da komplexe **Lambdas** schwer zu debuggen sind und komplexe Zugriffsdeklarationen viele Probleme verursachen können

### 2.14.1 Initialisierung von Lambda Captures

C++14

Lambda-Funktionen bzw. -Captures können seit C++14 ebenfalls initialisiert werden. Im Beispiel des entsprechenden Proposals «initialized Lambda Captures» ist `r` eine Referenz auf `::x`, und `x` eine Kopie von `::x`. Durch die Lambda-Operation wird die Referenz `::x` zu `6` und das aus der Kopie resultierende `y` zu `7`.

Die Anpassungen zu Generic Lambda Expression erlauben nun in C++14 Lambdas mit `auto` als Type Specifier in der Parameterdeklaration.

```
int x = 4;
auto y = [&r = x, x = x + 1]()->int
{
    r += 2;
    return x + 2;
}();
```

### 2.14.2. Lambda Captures mit «this» Pointer

C++20

Die folgende Notation ist eine Verbesserung der Lesbarkeit des Codes. Bei der Implementierung von Lambda Ausdrücken im Kontext einer Klasse würde implizit der Typ im Capture eingeschlossen, je nach default (s.o.) als Referenz oder Kopie.

Der Ausdruck

```
auto myLambda = [=, this](/* ... */) {/* ... */};
```

ist erst seit C++20 zulässig. Per Definition ist dies nun gleichbedeutend mit dem impliziten Capture als Kopie.

## 2.15 KONSTANTE AUSDRÜCKE

C++11

C++11 führt das Keyword `constexpr` ein, um die Limitation für konstante Ausdrücke von C++98 zu beheben. Das `constexpr`-Keyword ermöglicht die Evaluation eines Funktionswertes zur Übersetzungszeit. Seit C++17 ist `constexpr` auch in if-statements erlaubt, wodurch Code bereits zur Compilezeit ausgewertet und optimiert werden kann.

Regeln für `constexpr`

- Der Rückgabotyp darf nicht `void` sein
- In einer `constexpr`-Funktion dürfen keine Variablen deklariert und keine neuen Typen definiert werden
- Die Parameter einer Funktion, die für einen `return`-Ausdruck verwendet werden, müssen so «einfach» sein, dass der Ausdruck `constexpr` bleibt

**Beispiele:**

```
constexpr double pi{
    3.1416}; // const floating point instead of a #define PI 3.1416
constexpr int square(int x) { return x * x; }

// constant expressions and functions can be used in arrays and
similar
float fArray[square(5)];

// constant expressions can be used as template arguments
std::array<int, square(10)> arr;
struct Dummy {
    constexpr Dummy(int a, int b) : m_a(a), m_b(b) {}
    int m_a, m_b;
};

// fails because 'new' is a non-const operation
constexpr Dummy *construct(int x, int y) { return new Dummy(x, y); }

constexpr Dummy d{3, 4};
// d.m_a is transformed to a constant expression 3
int iArray[d.m_a];
```

```
// Here constexpr is used to facilitate template specialisation 'in-place'
template <typename T> class NameByType {
public:
    std::string to_string() {
        if
            constexpr(std::is_pointer<T>::value) { return "Pointer"; }
        else if
            constexpr(std::is_integral<T>::value) { return "Integral"; }
        else if
            constexpr(std::is_same<T, float>::value) { return "Float"; }
        else {
            return "Unknown";
        }
    }
};

// A fixed size storage class that can return different types by using auto and
// constexpr if (only available since C++17)
class MixedStorage {
public:
    template <std::size_t N> auto access() {
        if
            constexpr(N == 0) { return a; } // int
        if
            constexpr(N == 1) { return b; } // char
        if
            constexpr(N == 2) { return c; } // string
        if
            constexpr(N == 3) { return d; } // vector
        else if
            // return a fixed size array of size N
            constexpr(N > 3) { // Condition needs explicit specification or else
                // return type cannot be deduced correctly
                std::array<int, N> r;
            }
    }
};
```

```

        return std::move(r);
    }

    else {
        return 123;
    }
}
private:
int a = 1000;
char b = 64;
std::string c{"Hello world"};
std::vector<char> d{'a', 'b', 'c', 'd'};
};

```

### 2.15.1. Lambdaausdrücke mit constexpr & mutable C++17

Seit C++17 können auch Lambda-Funktionen mit **constexpr** markiert werden. Falls der Call-Operator der Funktion alle Voraussetzungen für einen konstanten Ausdruck erfüllt, ist dies jedoch auch in Abwesenheit des Keywords **constexpr** der Fall, falls sowohl der Capture als auch die Funktionsparameter die Bedingungen für einen konstanten Ausdruck erfüllen. Das Gegenteil davon ist die Markierung mit **mutable**. Ist dies der Fall, kann der Funktionskörper die By-Copy-Parameter im Capture verändern und deren non-const-Funktionen aufrufen.

### Beispiel:

```

class MutableClass {
public:
    int &non_const_a() { return _a; }
    const int &const_a() const { return _a; }

private:
    int _a{1000};
};

constexpr int c{1000};
// explicitly constexpr lambda
const auto constexpr_func = [&c](int v) constexpr { return c + v; };

// implicitly constexpr, no warning if no longer constexpr
const auto const_func = [&c](int v) { return c + v; };

MutableClass obj;
// an explicitly non-constexpr lambda that takes obj by value and modifies it
// Needs mutable keyword to access non_const_a()
auto non_const_func = [obj](int v) mutable { obj.non_const_a() += v; };

// if obj is passed by reference the mutable keyword is no longer needed
auto non_const_func_with_ref = [&obj](int v) { obj.non_const_a() += v; };

```

### 2.15.2. Initialisierung mit `constexpr`

C++20

Die Initialisierung globaler statischer Variablen zur Compile-Zeit wird mit C++20 abgesichert durch das Keyword `constexpr`. Es ist daher nur für Ausdrücke definiert, die zur Compile-Zeit ausgewertet werden können, und es wird garantiert, dass die Initialisierung nicht erst zur Laufzeit durchgeführt wird (vgl. `static`). Im Gegensatz zu `constexpr` wird keine konstante Destruktion garantiert, und `constexpr` kann nicht auf lokale Variablen angewendet werden. Die Verwendung von `constexpr` impliziert `constexpr`, aber nicht umgekehrt.

### Beispiel:

```
// OK, initialized at compile-time
constexpr auto GlobalPhrase = "Work's fine";

struct Dummy {
    constexpr static int MagicNumber;
};

// OK
constexpr int Dummy::MagicNumber = 42;

void local_function() {
    // Error: local variable cannot be declared
    constexpr auto tmp = 0;
}

const char * g() {
    return "dynamic initialization";
}
constexpr const char * f(bool p) {
    return p? "constant initializer" : g();
}

// OK, literal as constexpr
constexpr const char * c = f(true);

// Error: does not have a constant initializer
constexpr const char * d = f(false);
```

### 2.15.3. Immediate Funktionen mit `constexpr`

C++20

Als **Immediate Function** werden Funktionen bezeichnet, die forcieren, dass jeder Aufruf in einer `constexpr` resultiert, also in einer Compile-Zeit konstanten ausgedrückt werden kann. Damit können beispielsweise Präprozessor-Makros eliminiert werden. Für eine `constexpr` Funktion müssen daher alle Bedingungen von `constexpr` Funktionen erfüllt sein. Konstruktoren können `constexpr` sein, Destruktoren hingegen nicht.

#### Beispiel:

```
// Immediate Function, requires compile-time constants
constexpr int square(int n) {
    return n*n;
}

// OK
int square1 = square(10);

// OK
constexpr int constexpr_ten = 10;
int square2 = square constexpr_ten);

// OK
const int const_ten = 10;
int square3 = square(const_ten);

// Error: call to square is not a constant expression
int ten = 10;
int square4 = square(ten);

constexpr int const_init_fourty_two = 42;
const int const_fourty_two = 42;
struct CompileTimeInt {
    constexpr CompileTimeInt(int v) : value(v) {}
    const int value;
};
// Error: call to CompileTimeInt is not a constant expression
CompileTimeInt FortyTwo(const_init_fourty_two);
CompileTimeInt FortyTwo(const_fourty_two); // OK
```



## 2.16 ALIGNMENT

C++11

In C++11 kann das Memory-Alignment eines Objekttyps gelesen und gesteuert werden. Das Alignment eines Objekts gibt an, wie viele Bytes zwischen den Adressen von zwei hintereinanderliegenden Objekten dieses Typs reserviert werden müssen.

### 2.16.1 Alignof Operator

Gibt einen Wert vom Typ `size_t` zurück mit der Anzahl Bytes, die für das Alignment dieses Typs oder des Objekts verwendet wird. Dies entspricht üblicherweise dem Alignment des grössten internen Elements.

**Syntax:** `alignof(Type)`

### Beispiel:

```
struct C {
    char a; // size: 1, alignment: 1, offset 0
    char b; // size: 1, alignment: 1, offset 1
};          // sizeof(C): 2, alignof(C): 1

struct I {
    int n; // size: 4, alignment: 4, offset 0
    char c; // size: 1, alignment: 1, offset 4
    // + three bytes padding because of alignment 4 of in n
}; // sizeof(I): 8, alignof(I): 4

struct S {
    int a; // size: 4, alignment: 4, offset 0
    int b; // size: 4, alignment: 4, offset 4
    int c; // size: 4, alignment: 4, offset 8
};          // sizeof(S): 12, alignof(S): 4

std::cout << "sizeof(C) = " << sizeof(C) << " alignof(C) = " << alignof(C)
    << std::endl;
std::cout << "sizeof(I) = " << sizeof(I) << " alignof(I) = " << alignof(I)
    << std::endl;
// > sizeof(C) = 2 alignof(C) = 1
// > sizeof(I) = 8 alignof(I) = 4

std::cout << "sizeof(S) = " << sizeof(S) << " alignof(S) = " << alignof(S)
    << std::endl;
std::cout << "offsetof(S,b) = " << offsetof(S, b)
    << " offsetof(S,c) = " << offsetof(S, c) << std::endl;
```

### 2.16.2 Alignas Operator

Setzt das Alignment des Objekts auf die stärkste angegebene Expression, die grösser ist als Null, ausser das natürliche Alignment des Objekttyps wäre grösser. Der Compiler platziert Objekte in der Regel optimal, aber in seltenen Fällen können durch `alignas()` Performanceverbesserungen erreicht werden (z.B. Objekt überlappt eine Cacheline oder Memorypage). Andere Einsatzzwecke sind z. B. das Abbilden von Hardwareabhängigkeiten.

**Syntax:** `alignas(expression)`

**Beispiel:**

```
struct X {
    int a;           // size: 4, alignment: 4, offset 0
    alignas(16) int b; // size: 4, alignment: 16, offset 4
    int c;           // size: 4, alignment: 4, offset 20
    // + 8 bytes padding because the forced alignment of b
}; // sizeof(X): 32, alignof(X): 16 (forced by alignas of b)
std::cout << "sizeof(X) = " << sizeof(X) << " alignof(X) = " << alignof(X)
    << std::endl;
std::cout << "offsetof(X,b) = " << offsetof(X, b)
    << " offsetof(X,c) = " << offsetof(X, c) << std::endl;
```

### 2.17 MOVE SEMANTICS

C++11

Das neue **Move-Semantics**-Konzept wurde hauptsächlich eingeführt, um unnötige Kopien von temporären Objekten zu vermeiden. Um die Move-Funktionalität in Klassen verwenden zu können, müssen der Move Constructor und der Move Assignment Operator definiert werden.

**Move Constructor:** Erzeugen eines neuen Objekts aus einem anderen heraus (`other`), dessen Daten entnommen werden sollen.

**Signatur:** `T::T(T&& other)`

**Move Assignment Operator:** Überschreiben eines bestehenden Objekts mit den Daten eines anderen (`other`), dessen Daten entnommen werden sollen.

**Signatur:** `T& T::operator=(T&& other)`

**Move Semantics** in Containern:

Die Move-Funktionalität ist in Standard-Library-Containern weit verbreitet. Das heisst, alle Standard-Container unterstützen den Move Constructor und den Move Assignment Operator.

**Beispiel:**

```
struct Dummy {
    Dummy() : data(nullptr) {}
    Dummy(int myInt) : data(new int) { *data = myInt; };

    ~Dummy() { delete data; }

    // Move constructor, Object can be moved without new allocation of data
    Dummy(Dummy &&other) : data(other.data) { other.data = nullptr; }
    // move-assignment. Note that internal deletion of data first
```

```

Dummy &operator=(Dummy &&other) {
    if (data)
        delete data;
    data = other.data;
    other.data = nullptr;
    return *this;
}

// prevent copying, because of the internal allocation of 'data'
Dummy(const Dummy &) = delete;
Dummy &operator=(const Dummy &) = delete;

int *data;
};

int main(int, char **) {

    // Examples of moveable objects used in standard containers

    std::vector<Dummy> vd1;

    // move a temporary object into the vector
    // Since C++17 this guaranteed to be a move, see 'guaranteed copy elision'
    vd1.push_back(Dummy(100));

    vd1.emplace_back(200); // emplace without the temporary object

    Dummy di1(300);
    // Move di1 into the vector. that 'di1' is no longer accessible after the call
    vd1.push_back(std::move(di1));

    std::vector<Dummy> vd2[3]; // three empty vectors in an array
    vd2[0].push_back(Dummy(500));
    vd2[1] = std::move(vd2[0]); // move the whole vector

```

```

std::swap(vd2[1], vd2[2]); // swap two vectors

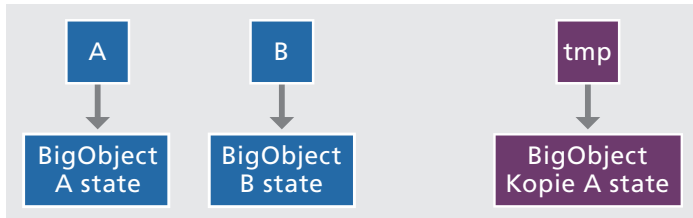
// retrieve an element. the vector does not change its size
std::vector<Dummy> result = std::move(vd2[2]);
}

```

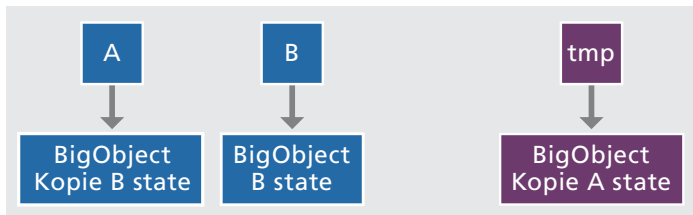
**Anwendungsbeispiel: Swap**

Klassische Variante (C++03):

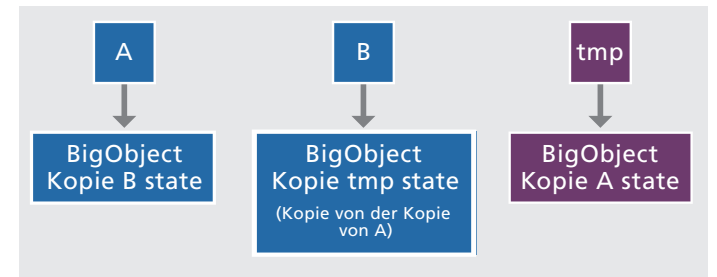
1. BigObject tmp, die erste Kopie



2. A kopiert den State von B, eine weitere Kopie



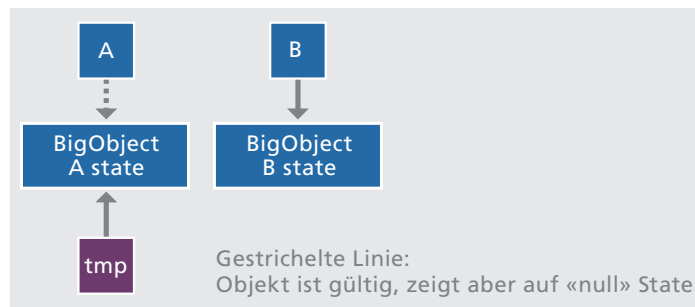
3. B kopiert tmp's Kopie (A's State), eine weitere Kopie



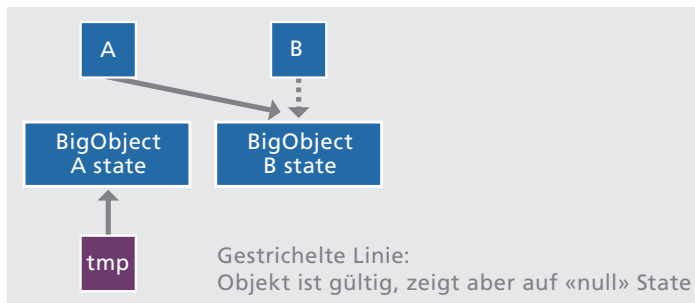
Alles in allem, drei Kopien für die Swap-Operation!

Variante mit C++11:

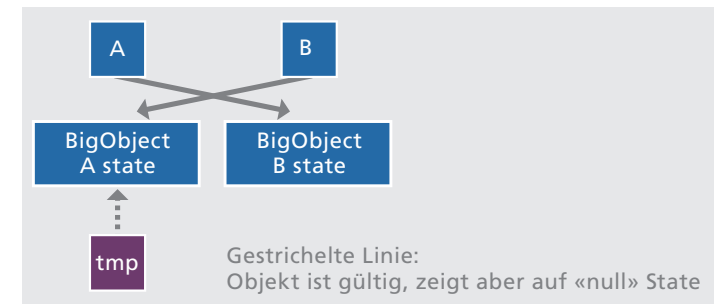
1. A's State nach tmp verschieben



2. B's State nach A verschieben



3. tmp's State nach B verschieben



Alles in allem, **KEINE** Kopien!

**Aber Achtung:** Im Prinzip ist es eine gefährliche Operation und kann dann verwendet werden, wenn der State nie mehr gebraucht wird. Es muss sichergestellt werden, dass die Variablen nicht referenziert sind!

Implementation der zuvor dargestellten Swap-Funktion mit explizitem Aufruf von `std::move` in C++11:

**Beispiel:**

```
// swaps two variables using move semantics
template <typename T> void swap(T &a, T &b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

## 2.18. COROUTINE

C++20

**Coroutines** sind Weiterentwicklungen von Funktionen, die die Ausführung temporär aussetzen und zu einem späteren Zeitpunkt wieder aufnehmen kann. Es werden keine Ressourcen blockiert, stattdessen wird der Ausführungszustand erhalten und beim nächsten Aufruf wiederhergestellt.

Anwendungsfälle sind vor allem Generatoren von unendlichen Datenströmen, Lazy-Evaluations oder Eventbasierte Applikationen. Durch **Coroutines** können auch asynchrone Anwendungen deutlich vereinfacht werden.

Dazu werden die Keywords `co_await`, `co_yield` und `co_return` eingeführt. Jede Funktion, die eines dieser Keywords benutzt, wird automatisch zur **Coroutine**.

**Syntax:** `co_await expression /`  
`co_yield expression /`  
`co_return expression`

**Coroutines** dürfen per Definition keine variadischen Argumente, keine `return` Anweisungen und keine Platzhalter-Rückgabewerte (`auto`) benutzen. **constexpr** Funktionen, Konstruktoren, Destruktoren und die main-Funktion dürfen keine **Coroutines** sein.

**Beispiel:**

```
// Number generator; creates an infinite stream
generator <int> get_integers(int start=0, int step=1) {
    for(int current = start;; current +=step)
        co_yield current;
}

// Wait for read from buffer
task<> read_buffer() {
    char data[1024];
    for(;;) {
        size_t n = co_await socket.read(buffer(data));
    }
}
```

Im Beispiel wird ersichtlich, dass weitere Typen definiert werden müssen, um **Coroutines** zu nutzen. Die Spezifikation im C++20 Standard entspricht mehr einer reinen Anleitung, wie **Coroutines**

und die dafür nötigen Typen implementiert werden können. Anwender sollten einsatzbereite Implementierungen aus Bibliotheken wie beispielsweise `cppcoro` verwenden, oder auf die Integration in C++23 setzen. Was der C++20 Standard definiert, sind die genannten Keywords und die Interfaces `Promise` und `Awaitable`.

Das `Promise` dient dazu, das Verhalten der `Coroutine` anzupassen, wenn sie aufgerufen oder beendet werden, bei (unbehandelten) Exceptions und die Ausgabe des eigentlichen Rückgabewertes. Es wird vom Compiler zum Erzeugen des `Promise-Handles` benötigt, welches zum Aus- und Fortsetzen der `Coroutine` verwendet wird.

Im `Awaitable` wird das Verhalten bei `co_await` angepasst: ob die `Coroutine` ausgesetzt wird, das Verhalten bei Fortsetzung der `Coroutine` etc. Zwei vom Standard bereits definierte `Awaitable` sind `suspend_always` und `suspend_never`, die die `Coroutine` entsprechend immer oder nie pausieren.

Für ein Beispiel zur Implementierung des `Promise` sei auf das `bbv` Repository ([https://github.com/bbvch/Cpp\\_Booklet](https://github.com/bbvch/Cpp_Booklet)) verwiesen.

## 2.19 VARIADIC TEMPLATES

C++11

Mit C++11 ist es neu möglich, Templates mit variabler Argumentanzahl zu erstellen. Solche Templates sind auch unter dem Namen `variadic Templates` bekannt. Dabei müssen nicht alle Argumente variabel sein, und es ist auch möglich, die Liste der Argumente leer zu lassen. Implementierungen von `variadic Templates` sind eher für die Erstellung von Bibliotheken gedacht.

**Syntax:** `template<typename ...Args>`

```
void aFunction(Args...args) { f(args...); }
```

`...Args` deklariert das Template `Parameter-Pack` (keine oder mehrere Argumente). Die « ... » dienen bei der Implementierung als Wiederholungsoperator des `Parameter-Packs`.

`...args` ist das Funktions-`Parameter-Pack` (keine oder mehrere Funktionsargumente).

`args...` ist das Entpacken des `Parameter-Packs` und ist eine der wenigen Operationen, die auf das `Parameter-Pack` angewendet werden können.

**Beispiele:**

```
// variadic template function definition
template <typename... Args> void aFunction(Args... args) {}
aFunction(42, 1.14159, "Hello", Dummy());
aFunction(1, 2, 3, 4, 5);
```

```
std::string reorder_and_concat() { return std::string(); } // End of recursion

// recursion popping the first template argument and concatenating
template <typename T, typename... Args>
std::string reorder_and_concat(T t, Args... args) {
    std::string to_be_sorted(t);
```

```

std::sort(to_be_sorted.begin(), to_be_sorted.end());
return to_be_sorted + reorder_and_concat(args...);
}
// returns "ABC"
std::cout << reorder_and_concat("CBA") << "\n";

// returns ABCJKL
std::cout << reorder_and_concat("CBA", "KLJ") << "\n";

// returns ABCJKLXYZ
std::cout << reorder_and_concat("CBA", std::string("KLJ"), "ZYX") << "\n";

```

```

// variadic template class definition
template <typename... Args> class aClass {};
aClass<std::string, int> ac1;
aClass<char, char, double, std::string> ac2;
aClass<> ac3;

```

### 2.19.1 Fold Expressions

C++17

Mit C++17 können Parameter-Packs für Variadic Templates mittels sogenannten **Fold Expressions** direkt angewandt werden, falls nur einfache Operatoren benötigt werden.

Unterstützt sind:

- boolesche Operatoren (&&, ||, <, >, ==, !=)
- mathematische Operatoren (+, -, \*, /, %)
- bitweise Operatoren (&, |, ^, >>, <<)
- Zuweisungsoperatoren auch in Kombination mit anderen Operatoren (=, /=, &=...)

Spezielle Operatoren sind zudem die Verknüpfungs- und Dereferenzierungsoperatoren, welche es erlauben, das Argument-Pack zu dereferenzieren, falls es sich um Pointer-Typen handelt

Ein weiterer spezieller Operator ist der Komma-Operator, der verwendet werden kann, um das Parameter-Pack an normale Funktionen weiterzuleiten.

**Fold Expressions** können sowohl links- als auch rechtsassoziativ geschrieben werden und unterstützen unäre und binäre Operationen.

```

template <typename... Args> auto binary_left_fold(Args &&... args) {
    return (10 + ... + args);
}

template <typename... Args> auto unary_left_fold(Args &&... args) {
    return (... - args);
}

template <typename... Args> auto unary_right_fold(Args &&... args) {
    return (args - ...);
}

std::cout << binary_left_fold(1) << "\n"; // 11 (10 + 1)
std::cout << binary_left_fold(1, 2, 3) << "\n"; // 16 (10 + 1 + 2 + 3)
// 115.499L (internal cast to double) (10 + 1 + 4.5f + 99.9999L)
std::cout << binary_left_fold(1, 4.5f, 99.9999L) << "\n";

std::cout << unary_left_fold(10, 3, 2) << "\n"; // (10 - 3) - 2 = 5
std::cout << unary_right_fold(10, 3, 2) << "\n"; // 10 - (3 - 2) = 9

```



## 2.20. CONCEPTS

C++20

Das Design von C++ basiert im Wesentlichen auf der Abstraktion von Klassen und Funktionen mit entweder spezifischen (z.B. `char`, `int`, `string`) oder generischen (Template-basierten) Datentypen. Daraus ergeben sich einerseits Probleme durch duplizierten Code, **Narrowing Conversion** und **Integral Promotion**. Andererseits fehlen mit Templates oft klare Anforderungen an die Datentypen und Probleme werden vom Compiler mit kryptischen Fehlermeldungen dokumentiert. Mit diesen Problemen räumen **Concepts** auf, indem explizit semantische Einschränkungen auf Template-Parameter definiert werden. Diese Einschränkungen werden zur Compile-Zeit überprüft.

**Concepts** ermöglichen es, konkrete Anforderungen und Einschränkungen an Datentypen zu formulieren. Damit stellen sie mächtige Werkzeuge zur Verfügung, um besser auszudrücken, welche Datentypen für Funktionen oder Klassen zugelassen sind. Das bedeutet, der Einsatz von Templates wird erleichtert, auch weil Fehlermeldungen vom Compiler deutlich verständlicher sind. Ein **Concept** besteht aus einem (oder mehreren verknüpften) konstanten Ausdruck.

**Syntax:** `template <template-parameter-list>`  
`concept concept-name =`  
`constexpr constraints;`

### Beispiel

```
// T must be const
template <typename T>
concept Constant = std::is_const<T>::value;

// T must have the same alignment as int
template <typename T>
concept Alignment_Int = std::alignment_of_v<T>
    == std::alignment_of<int>();

// T must be a signed integral
template<class T>
concept SignedIntegral = std::is_integral_v<T>
    && std::is_signed_v<T>;
```

Die Concepts können dann mit verschiedenen Notationen angewendet werden, die in ihrer Aussage identisch sind:

- Requires Constraint
- Trailing Requires Constraint
- Constrained Template Parameter.

Das entscheidende Keyword ist `requires`, welches den konstanten Ausdruck bzw. das Concept markiert. Alternativ kann die Template-Definition statt dem generischen `typename` direkt auf den Namen des Concept verweisen.

**Beispiel**

```
// Requires Constraint
template <typename Register>
    requires Alignment_Int< Register >
void CopyTo(Register & reg);

// Trailing Requires Constraint
template <typename Value>
void UpdateValue(Value& val) requires Constant<Value>;

// Constrained Template Parameter
template<SignedIntegral Position>
void MoveRelative(Position& target);
```

Concepts können ebenfalls als Ergänzung von `auto` verwendet werden. Indem das Concept direkt `auto` vorangestellt wird, sind Funktionen möglich, die sowohl als Argument als auch Rückgabe `auto` akzeptieren. Aus dem unspezifischen wird dadurch ein Typenbeschränkter Platzhalter.

```
// Abbreviated Function Template
SignedIntegral auto Clamp(SignedIntegral auto min,
SignedIntegral auto max, SignedIntegral auto val)
{
    return std::max(std::min(max, val), min);
}
```

Die **Concepts** erweitern den Einsatz der Sprache deutlich. Es können sehr konkrete Anforderungen und Einschränkungen an Klassen und Funktionen implementiert und entsprechend eingesetzt werden, um sicher zu stellen, dass nur passende Argumente übergeben werden. Es können klar definierte Schnittstellen defi-

niert und so Fehler früh vermieden werden. **C++** bietet dafür die **type\_traits** Bibliothek an, um die Definition von **Concepts** zu vereinfachen.

Der Standard liefert bereits einige vordefinierte **Concepts**, sodass bestehender Code einfacher als je zuvor abstrahiert und vor allem auch expliziter in seinen Anforderungen gegenüber Datentypen umgesetzt werden kann. Sie ermöglichen ausserdem viele weitere neue Features wie Ranges, oder den Dreiwegevergleichs-Operator.

```
// #include <compare>
std::three_way_comparable

// #include <concepts>
std::derived_from
std::movable
std::copyable
std::swappable

// #include <ranges>
std::sortable
std::mergeable
```

## 2.21 RETURN TYPE DEDUCTION & DECLTYPE (AUTO)

C++14

Die C++14-Erweiterung der **Return Type Deduction** wirkt sich auf **decltype(auto)** aus. **Return Type Deduction** tritt auf, wenn eine Funktion ihren Returnwert als **auto** definiert und der Compiler dann den passenden Typ aus der Funktionsimplementierung finden muss.

```
template <typename T, typename U> auto add(T t, U u) {
    // return type deduced from operator+(T, U)
    return t + u;
}
```

Zu Problemen kann es bei Verwendung von Rekursion kommen, oder wenn tatsächlich verschiedene Rückgabetypen möglich sind. Returntyp **auto** deduziert niemals auf eine Referenz, die Universalreferenz **auto&&** jedoch immer. Die Änderung in C++14 richtet sich hier an **decltype(auto)**, welches bspw. implizit auch bei Klammerung des Returnwertes zu einer Referenz führt.

### Beispiel:

```
// C++11 Version
template <typename Container, typename Index>
auto assignValue11(Container &&cont, Index idx)
    -> decltype(std::forward<Container>(cont)[idx]) {

    return std::forward<Container>(cont)[idx];
}

// C++14 Version
template <typename Container, typename Index>
decltype(auto) assignValue14(Container &&cont, Index idx) {

    return std::forward<Container>(cont)[idx];
}

// this allows the following initialisation,
// note the missing template arguments
std::vector<int> vec;
assignValue14(std::vector<int>(), 1) = 10;
```

## 2.22 STRUCTURED BINDINGS

C++17

Mit C++17 wurde das vereinfachte Entpacken von Strukturen mit zur Compiletime bekannter Grösse eingeführt. Die zu entpackenden Daten müssen dabei nicht zwingend vom selben Typ sein.

```
struct Packed {
    int x;
    char y;
    float z;
};

class cls {
public:
    int m;
    float n;
};

auto tuple = std::make_tuple(1, 'a', 2.3);
std::array<int, 3> a{1, 2, 3};
Packed p;
cls cl;

// unpack the tuple into individual variables declared above
const auto[i, c, d] = tuple;
// also works with references
auto & [ ref_i, ref_c, ref_d ] = tuple;

// fixed size array also works
auto[j, k, l] = a;

// structs and classes can also be bound
auto & [ x, y, z ] = p;
auto[m, n] = cl;

// extracting key & value at the same time from a map
```

```
std::map<int, float> map;
for (auto && [ k, v ] : map) {
    // do something
}
```

## 2.23 SELECTION STATEMENTS MIT INITIALIZER

C++17

Seit C++17 können Verzweigungen (If, Switch) mit Anweisungen zur Initialisierung versehen werden. Dies hilft Variablen explizit auf einen kleineren Scope zu begrenzen.

**Syntax:** if(initialization; condition)  
bzw: switch(initialization; condition)

```
// initialize i rand (); then perform the check the if
if (int i = std::rand(); i % 2 == 0) {
    std::cout << "i is even" << std::endl;
} else {
    std::cout << "i is odd" << std::endl;
}

// initialize i with rand(); then perform the switch
switch (int i = std::rand(); i % 3) {
case 0:
    std::cout << "i is a multiple of 3" << std::endl;
    break;
case 1:
    std::cout << "i is not a multiple of 3" << std::endl;
    break;
case 2:
    std::cout << "i is almost a multiple of 3" << std::endl;
    break;
}
```

## 2.24 STANDARDATTRIBUTE UND ANNOTATIONS

C++14

C++17

Verschiedene Standardattribute wurden seit der Einführung von C++11 hinzugefügt. Seit C++17 können Attribute für Namespaces und Enums verwendet werden, dies macht insbesondere bei `[[deprecated]]` Sinn. Es können mehrere Attribute miteinander spezifiziert werden.

### `[[noreturn]]`

C++17

`[[noreturn]]` zeigt an, dass die Funktion nicht zurückkehrt und das Programm in einem undefinierten Zustand ist, falls dies trotzdem geschieht. Dies ist üblicherweise der Fall bei Funktionen, die zwingend eine Exception werfen, bei Funktionen, die einen **programm-exit** vornehmen oder bei Endlosschleifen.

```
// indicates that this function never returns
[[noreturn]] void always_throw() { throw 123; };
```

### `[[deprecated]]`

C++14

`[[deprecated]]` oder `[[deprecated(«reason»)]]` zeigt an, dass eine Funktion nicht mehr benutzt werden sollte, optional durch Angabe eines Grundes. Ist ein Symbol einmal als **deprecated** markiert, lässt sich diese Markierung innerhalb der **compile-unit** nicht mehr aufheben. Wird ein als **deprecated** markiertes Symbol verwendet, wird eine Warnung vom Compiler ausgegeben.

```
// generates a compiler warning if called
[[deprecated("black magic is no longer used")] void black_magic(){};

// generates a compiler warning if called
[[deprecated]] void ancient_magic() {}
```

### `[[fallthrough]]`

C++17

`[[fallthrough]]` wird in einem switch-Statement als eigene Zeile unmittelbar vor einem case-Statement verwendet. Das Attribut zeigt an, dass absichtlich kein `break`; gesetzt wurde und der Compiler keine Warnung anzeigen soll. Folgen mehrere case-Statements unmittelbar aufeinander ohne Code dazwischen, wird automatisch `[[fallthrough]]` angenommen.

```
void fall_through(int i) {
    switch (i) {
        case 0:
        case 1:
        case 2: // until here implicit [[fallthrough]]
            foo();
            [[fallthrough]]; // don't forget the semicolon
        case 3:
            bar();
        default: // compiler might issue a warning, depending on the flags set
            break;
    }
}
```

**[[nodiscard]]**

C++17

`[[nodiscard]]` kann auf die Deklaration von Enums oder von Datenstrukturen angewendet werden. So zeigt der Compiler eine Warnung an, falls der Rückgabewert einer Funktion dieses Typs unmittelbar nach dem Aufruf nicht mehr weiterverwendet wird. Wird eine Funktion mit `[[nodiscard]]` markiert, darf der Rückgabewert nicht verworfen werden, unabhängig vom Typ.

```
struct[[nodiscard]] demon{}; // Demons need to be kept and named
struct ghost {};           // Ghosts can be free

demon summon_demon() { return demon(); }

// summoned ghosts need to be kept
[[nodiscard]] ghost summon_ghost() { return ghost(); }

void summon() {
    auto d = summon_demon(); // OK
    auto g = summon_ghost(); // OK

    // Compiler Warning, because returned demon is not kept
    summon_demon();

    // Compiler Warning, because the function of summoning is nodiscard
    summon_ghost();}

```

**[[maybe\_unused]]**

C++17

`[[maybe_unused]]` unterdrückt Compiler-Warnungen für Variablen, **typedefs** und nicht statische Data Members einer Klasse, falls diese nicht verwendet werden. Häufig ist dies der Fall bei Hilfsvariablen im Zusammenhang mit Debug-Code, der im Release nicht mehr dazu kompiliert wird wie z. B. Asserts.

```
// omit compiler warning if thing1 on thing2 is not used
[[maybe_unused]] void f([[maybe_unused]] bool thing1,
                        [[maybe_unused]] bool thing2) {
    [[maybe_unused]] bool b = thing1 && thing2;

    assert(b); // in release mode, assert is compiled out, and b is unused
               // no warning because it is declared [[maybe_unused]]
    assert(thing1 &&
           thing2); // parameters thing1 and thing2 are not used, no warning
}

```

**[[carries\_dependency]]**

C++17

`[[carries_dependency]]` zeigt im Zusammenhang mit `std::atomic` an, dass eine Abhängigkeitskette eines **Release Consume Memory Order** durch eine Funktion hindurchpropagiert wird. Dies erlaubt dem Compiler, eine Funktion als «transparent», was das Memory Ordering betrifft, zu behandeln, ohne dass der Code inline-compiled sein muss. Allein die Anwesenheit des Attributes garantiert jedoch nicht, dass der Code in der Funktion auch tatsächlich die Bedingungen für eine solch transparente Behandlung der Memory-Struktur erfüllt.

```

void opaque_func(int *p){/* do something with p */};

[[carries_dependency]] void transparent_func(int *p) {
    /* do something with p */
}

void illustrate_carries_dependency() {
    std::atomic<int *> p;
    int *atomic = p.load(std::memory_order_consume);

    if (atomic)
        std::cout << *atomic << std::endl; // transparent for the
                                             compiler

    if (atomic)
        opaque_func(atomic); // if from another compile unit and not
                              inline the
                              // compiler might construct a memory fence
                              here

    if (atomic)
        transparent_func(atomic); // marked as to work in the same
                                   memory-dependency
                                   // tree, compiler can omit the memory
                                   fence}

```

**[[likely]] / [[unlikely]]**

C++20

Die Attribute `[[likely]]` / `[[unlikely]]` sind Vorhersagen, die an bedingte Zweige platziert werden können, um dem Compiler zu signalisieren, dass die Bedingungen sehr wahrscheinlich respektive sehr unwahrscheinlich erfüllt werden. Dadurch können Optimierungen so vorgenommen werden, dass die Anzahl der Prozessor-Zyklen zur Ausführung minimiert werden kann.

Für den Fall, dass die Vorhersage fehlschlägt, wird die Ausführung hingegen aufwändiger. Deshalb sollten die Vorhersagen nur für Zustände getroffen werden, die mit einer hohen Häufigkeit gegenüber anderen eintreffen, beispielsweise die Überprüfung eines Pointers, der nur in seltenen Ausnahmefällen ungültig ist.

```

//    Called very often, with a valid pointer
void copy_from_buffer(char * buffer, size_t size)
{
    if(nullptr == buffer) [[unlikely]]
    {
        . . .
    }
    . . .
}

void check_result(ErrorCode error)
{
    switch(error)
    case ErrorCode::NoError: [[likely]]
        break;
    // ...
}

```

## 2.25 EINFÜHRUNG WEITERER SYNTAX-ERWEITERUNGEN

### 2.25.1. User Defined Literals

C++20

Als **User Defined Literals** werden vom Nutzer definierte Symbole bezeichnet, denen explizite Werte zugewiesen sind. Dies können beispielsweise Einheiten oder Bezeichner sein. Die Sprache kann somit erweitert und angepasst und nicht zuletzt Fehler durch mehrdeutige Interpretationen vermieden werden. Zum Vergleich seien hier die vordefinierten Literale, wie z.B. **u** (**unsigned**), **ull** (**unsigned long long**) oder **f** (**float**) erwähnt. Diese Funktion ist für integrale Typen wie **integer**, **float**, **char**, **bool** etc. vorbehalten.

**Syntax:** <ReturnType> operator"" <Suffix>(<Parameters>)

```
// given a class Length
Length operator "" _km(unsigned long long n) {
    return Length{n, Length::Kilometers};
}

Length operator "" _mi(unsigned long long n) {
    return Length{n, Length::Miles};
}

const auto length = 23_mi + 42_km;
```

### 2.25.2. Binary Literals

C++14

Verschiedene kleinere Erweiterungen, die in C++14 hinzukommen, ermöglichen eine komfortablere Benutzung der Sprache.

**Binary Literals** erlauben nun, eine Zahl als binär zu kennzeichnen, genau gleich dem **0x** Prefix für hexadezimale Schreibweise.

```
//binary literal
int i = 0b1000;
```

Seit C++17 können auch **float** und **double** als hexadezimale Zahlen geschrieben werden. Bei hexadezimalen Zahlen muss zwingend der Exponent spezifiziert werden. Der Exponent wird jeweils als  $2^x$  angegeben.

```
// implicitly cast to float
float f = 0x1.2p3; // = decimal 1.125 or 1. + 1/16 * 2 * 2^3;

// with f (or F) suffix to explicitly state it as a float
float f1 = 0xA.1p3f;

// = decimal 0.5 omitting value in front of decimal separator
float f2 = 0x.1p3;

// = decimal 80 omitting value after decimal separator
float f3 = 0xA.p3;

double d = 0xDEAD.D0D0p2;
double d2 = 0xDEAD.D0D0p21; // explicitly stating double
```

Hochkommata werden nun vom Compiler als Digit-Separatoren interpretiert.



**Beispiel:** Single Quotation Mark as Digit Separator, die Stelle der Hochkommata ist dabei irrelevant, d. h., `1'2'22'333` kompiliert genauso in C++14.

```
// single quotation mark as digit separator. Has no effect on
// the interpretation
int x = 1'222'333;
```

## 2.26 LOCKERUNGEN KONTEXT BEZOGENER C++-UMWANDLUNGEN C++14

Implizite Umwandlung bei `new`, `delete`, `array-bounds` und `switch` sind neu erlaubt. Literale können zu `constexpr` umgewandelt werden. Einige Spezialfälle werden dadurch nun besser geregelt, ansonsten ist die Erweiterung minimal.

### Sizeof Operator C++11

Der `sizeof` Operator funktioniert bei C++11 neu auch für Klassen-Members.

**Syntax:** `sizeof(expression)`

## 2.27. MODULES C++20

Das Prinzip, Code in Header und Source Files zu teilen, ist sicherlich durch den Aufwand und die Konflikte mit der **One Definition Rule (ODR)** eines der grössten Hemmnisse beim Lernen und Anwenden von C++. Das Inkludieren eines Header-Files entspricht dem textuellen Einfügen. Bei kleinen Änderungen im Header müssen alle abhängigen Translation Units neu kompiliert werden, was vor allem beim Einsatz von Templates sehr aufwendig wird. Allein die Reihenfolge von Includes kann entscheiden, ob der Build korrekt ist.

**Module** werden generiert aus dem Source-File, welches zum Export als Modul gekennzeichnet wird, und können separat kompiliert werden. Sie ermöglichen es, Symbole zu exportieren und nur diejenigen zu verwenden, die wirklich gebraucht werden. Der Aufwand, Deklaration und Definition konsistent zu halten entfällt, da Header-Files nicht mehr benötigt werden. Das Erstellen von Anwendungen wird unabhängiger vom Präprozessor.

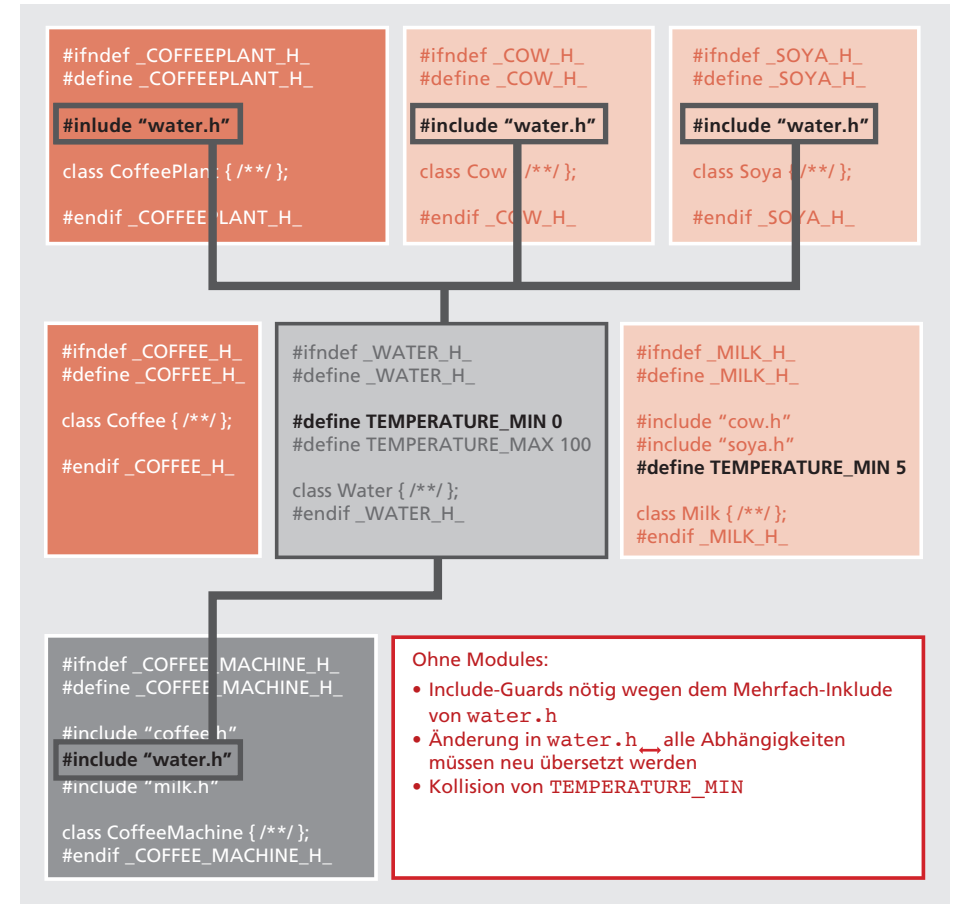
**Code** kann in Module gebündelt und Abhängigkeiten minimiert werden. Statt Header Files mit dem Präprozessor textuell einzufügen, können nun Module oder Submodule importiert werden. Mehrfach-Includes, Include-Reihenfolgen, Include-Rückkopplungen, Präprozessor-Überprüfungen unter anderem werden dadurch überflüssig. Die totale Compile-Zeit wird verringert und inkrementelle Builds vereinfacht.

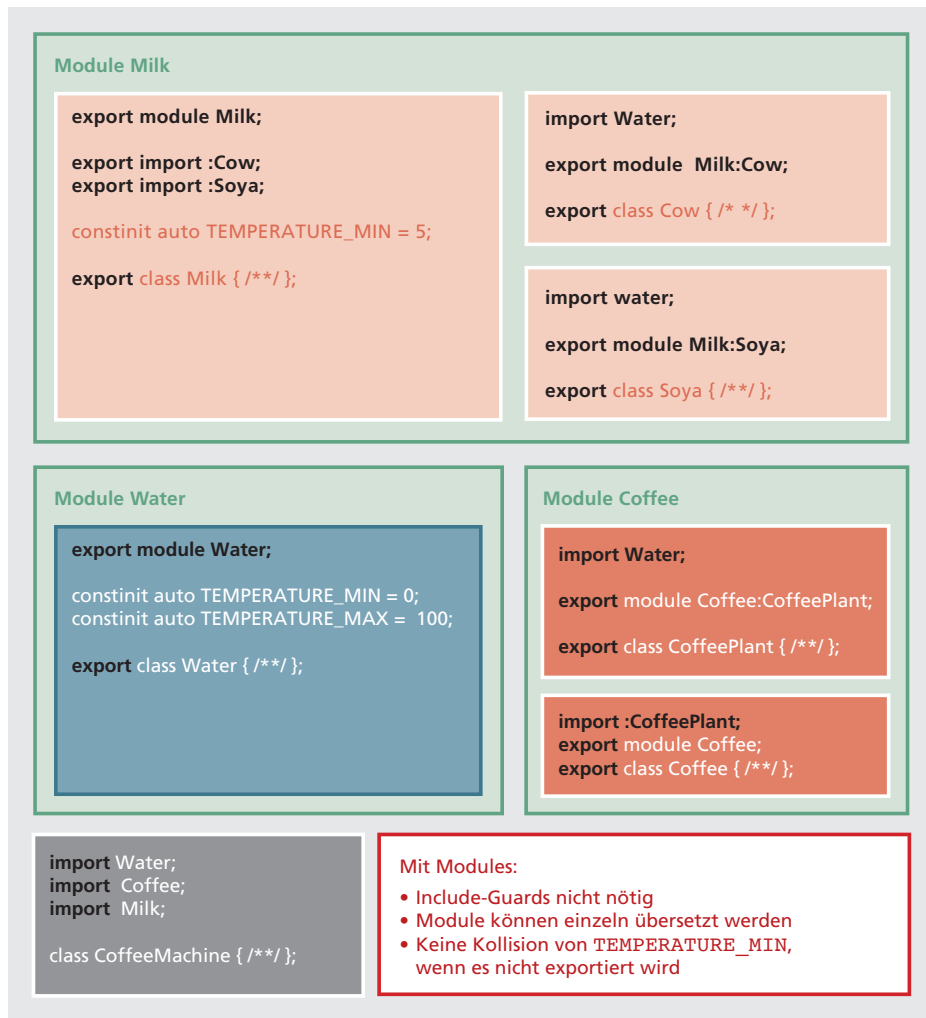
Um Module zu erstellen, wird das Keyword `«export»` (ohne `'#'`) für jedes einzelne Symbol verwendet, welches von anderen genutzt werden könnte. Die Nutzung erfolgt über das Keyword `«import»` (ohne `'#'`) und es gibt nur wenige Regeln zu beachten.

Es darf beispielsweise vor dem Import keine Deklarationen geben, und es darf weiterhin keine gegenseitigen Interface-Abhängigkeiten und somit keine wechselseitigen Imports geben.

Der Linker bleibt von diesem Konzept unbeeinträchtigt, d.h. es werden weiterhin Objekt-Files oder Bibliotheken benötigt.

Die Grenzen des aktuellen Standards sind vor allem die Abwärts-Kompatibilität. Zum anderen bringen **Modules** nicht automatisch Namespaces (vgl. Python), sodass sich weiterhin Konflikte bei Symbolen ergeben können. Der Austausch von Modules als Binaries wird weiterhin nur innerhalb der Einschränkung derselben Architektur, Compiler-Version etc. möglich sein.





## 2.28 \_\_HAS\_INCLUDE PRÄPROZESSOR

C++17

### DIREKTIVE

Seit C++17 kann bereits zur Compilezeit die Verfügbarkeit eines Headers evaluiert werden. Dies erleichtert das Schreiben von Plattform- oder Compiler abhängigem Code.

```

#if __has_include(<unistd.h>)
#define OPEN_SHARED dlopen
#elif __has_include(<windows.h>)
#define OPEN_SHARED LoadLibrary
#else#pragma error("loading shared libraries not supported");

```

### 2.28.1 Guaranteed Copy elision

C++17

Vor C++17 war es für Compiler optional, ob bei Funktionen, die Objekte by Value zurückgeben, Copy- bzw. Move-Konstruktoren aufgerufen werden oder nicht (Return Value Optimization). Mit dem neuen Standard ist eine solche Optimierung zwingend, z. B. das Schreiben von Factory-Methoden auch für Klassen mit gelöschten Konstruktoren.

```
class A
{public:
  A() = default;
  A(const A &rhs) = delete;
  A(const A &&rhs) = delete;
  ~A() = default;
};

// Without elision this is illegal, as it performs a copy/move of A which has
// deleted copy/move ctors
A f() { return A{}; }
// OK, because of copy elision. Copy/Move constructing an anonymous A is not
// necessary
A a = f();
```

Nach Klassen- und Funktionstemplates bekommt C++14 nun auch **Variablentemplates**. Konstanten können somit bspw. mal als Zahl für Berechnungen und mal direkt als String behandelt werden. Ein manchmal nicht ganz einleuchtendes Beispiel benutzt die Zahl Pi.

### Beispiel:

```
template<typename T>
constexpr T pi = T(3.1415926535897932385);

auto two_pi = pi<float> * 2;
```

### 2.29 ANPASSUNG ZUR CODEOPTIMIERUNG

C++14

Die **Clarifying Memory Allocations** erlauben dem Compiler, beim Allozieren/Dealozieren mehrere Aufrufe zusammenzufassen, sofern die maximale Grösse dadurch nicht zunimmt.

Bei der **Sized Deallocation** wird es in C++14 nun möglich, dem **delete**-Operator einen Parameter mitzugeben. Mit C++11 können Programmierer einen statischen Memberfunktions-Operator **delete** definieren, der einen Grösseparameter nimmt, der die Grösse des zu löschenden Objekts angibt. Der äquivalente globale Operator **delete** ist in C++11 noch nicht verfügbar. Dieses Fehlen hat negative Konsequenzen für die Performance. Moderne Memory-Allokatoren allozieren oft in Grössenklassen und speichern aus Speicherplatzeffizienzgründen nicht die Grösse eines Objekts neben dem Objekt selbst. Die Deallokation erfordert dann die Suche nach dem Platz der abgelegten Grössenklasse, die das Objekt beschreibt. Diese Suche kann teuer sein, insbesondere da sich die gesuchten Datenstrukturen oft nicht im Cache-Memory befinden. Auch hier geht es um Umformulierungen einiger Passagen des C++11-Standards. Auch diese Änderung betrifft kaum einen Entwickler. Der interessierte Leser sei an den ISO-Standard auf [isocpp.org](http://isocpp.org) verwiesen.

## 3 STANDARD-LIBRARY-ERWEITERUNGEN

Da es den Rahmen des Booklets sprengen würde (siehe Einleitung), kann hier auf die Standard-Library-Erweiterung nicht eingegangen werden. Zu diesem Thema gibt es Bücher und Online-Informationen. Weitergehende Informationen finden Sie zum Beispiel im Buch «The C++ Standard Library Second Edition» [3] oder auf der Webseite <http://en.cppreference.com>.

## 4 FAZIT

Die Sprache C++ bleibt weiterhin eine sehr anspruchsvoll zu erlernende Sprache, die viel Geduld und Ehrgeiz erfordert. Trotzdem ist sie wichtiger und populärer als jemals zuvor. Die grössten Meilensteine sind bisher der Standard von 2011 und 2020, gemessen am Umfang der eingereichten Beiträge und der veröffentlichten Features. Die Werkzeuge, die dem Entwickler hiermit zur Verfügung stehen, bilden die Grundlage für die Lösung komplexester Problemstellungen.

Die Werkzeuge sollten nach je Bedarf sinnvoll eingesetzt werden. Einen Überblick hierfür bildet dieses Booklet.

Und was bringen die neuen Standards? Zum einen ist C++ typensicherer geworden und Limitationen der früheren Standards wurden behoben. Zum anderen wurde die Sprache mit neuen Funktionen erweitert, welche bei anderen Sprachen bereits umgesetzt sind. Um ein paar zu nennen: vereinheitlichte Initialisierung, Multithreading-Unterstützung, allgemeine Programmierunterstützung und Performanceverbesserungen. Wie die Kernsprache wurde auch die Standard Library überarbeitet, erweitert und verbessert. Alles in allem sind die neuen Standards weitere Schritte weiter weg von «C mit Klassen» hin zu einer modernen und starken Programmiersprache. Im Intervall von 3 Jahren werden seit 2011 neue Sprachstandards veröffentlicht, und die steigende Anzahl der Beiträge ist ein deutliches Zeichen, wie wichtig und lebendig C++ weiterhin ist.

Wir denken, der neue Standard ist eine gelungene Erweiterung, die dem Softwareentwickler bei seiner täglichen Arbeit hilft, besseren und effizienteren Code zu entwickeln.

## 5 ANHANG

### 5.1 AUTOREN

Dieses Booklet wurde durch das Kompetenzzentrum C++ der bbv erarbeitet. Ganz besonders möchten wir uns bei Alain Baumeler für die erste Fassung zum C++11-Standard bedanken, ohne seine Grundarbeit wäre dieses Booklet nie zum Druck gelangt.

Für die Updates auf C++14 und 17 möchten wir den folgenden Personen danken. Silvan Wegmann, Thomas Meister, Jürgen Messerer, Raphael Meyer, Michel Estermann, Lothar Rubusch und Dominik Berner. Weiter möchten wir dem Korrektorat der bbv und Michael Zentner für die visuelle Gestaltung danken.

Das Update auf C++20 entstand unter der Mitarbeit von Christian Fischer, Dominik Berner, Silvan Wegmann et al.

### 5.2 HINWEIS ZUR KOMPILIERFÄHIGKEIT DER BEISPIELE

Alle angegebenen Codebeispiele wurden sorgfältig entwickelt und mit allen gängigen Compilern (Microsoft Visual Studio, clang, g++) auf dem jeweils aktuellen Stand getestet. Einige Features aus dem Standard ISO/IEC 14882:2020 sind noch nicht einheitlich in den Releases der genannten Compiler implementiert, sodass sich hier Unterschiede ergeben können.

Sollten sich trotz Überprüfung auf korrekte Syntax Fehler eingeschlichen haben, bitten wir dies zu entschuldigen.

Alle Beispiele sind hier verfügbar:

[https://github.com/bbvch/Cpp\\_Booklet](https://github.com/bbvch/Cpp_Booklet)

### 5.3 QUELLENVERZEICHNIS

C++ ISO Standards von <https://isocpp.org>

C++ Reference <http://en.cppreference.com>.

Wikipedia <http://www.wikipedia.org>.

C++11 programmieren, Torsten T. Will

ISBN: 978-3-8362-1732-3

Effective Modern C++, Scott Meyers

ISBN: 978-1-491-900399-5

C++ Standards Support in GCC <https://gcc.gnu.org/projects/cxx-status.html>

Barteks Coding Blog

<http://www.bfilipek.com/2017/01/cpp17features.html>

Simon Brands Blog

<https://blog.tartanllama.xyz/spaceship-operator>

Dawid Pilarski

<https://blog.panicsoftware.com/your-first-coroutine>

Hannes Hedewell Blog

<https://hannes.hauswedell.net/>

<https://cor3ntin.github.io/posts/span/>

[http://www.stroustrup.com/good\\_concepts.pdf](http://www.stroustrup.com/good_concepts.pdf)

clang documentation

<https://clang.llvm.org/docs/>

cppcoro library for coroutines

<https://github.com/lewissbaker/cppcoro>

## 5.4 LITERATUR

[1] C++11, Der Leitfaden für Programmierer zum neuen Standard, Rainer Grimm, ISBN: 978-3-8273-3088-8

[2] C++11 programmieren, Torsten T. Will,

ISBN: 978-3-8362-1732-3

[3] The C++ Standard Library Second Edition, Nicolai M. Josuttis, ISBN: 978-0-321-62321-8

[4] Presentation Materials: Overview of the New C++ (C++11), Scott Meyers [PDF] [http://www.artima.com/shop/overview\\_of\\_the\\_new\\_cpp/](http://www.artima.com/shop/overview_of_the_new_cpp/)

[5] The C++ Programming Language (4th Edition),

Bjarne Stroustrup, ISBN: 978-0321563842





bbv Software Services AG ist ein Schweizer Software- und Beratungsunternehmen, das Kunden bei der Realisierung ihrer Visionen und Projekte unterstützt. Wir entwickeln individuelle Softwarelösungen und begleiten Kunden mit fundierter Beratung, erstklassigem Software Engineering und langjähriger Branchenerfahrung auf dem Weg zur erfolgreichen Lösung.

Unsere Booklets und vieles mehr finden Sie unter  
[www.bbv.ch/publikationen](http://www.bbv.ch/publikationen)

**MAKING VISIONS WORK.**

[www.bbv.ch](http://www.bbv.ch) · [info@bbv.ch](mailto:info@bbv.ch)