



Typen und Value-Kategorien in C++

Mash up of Meeting C++ 2018

Gabriel Nützi, 2018 at Cyfex AG, Zurich

[[Teil 1](#)] [[Teil 1 Bemerkungen](#)] [[Teil 2](#)]

[[Git Repo](#)]



Über Statistik

15-50 Fehler per 1000 Zeilen geliefertem Code

[unabh. der Sprache, Steve McConnell, *Code Complete*]



Über `constexpr`

if `constexpr` must not introduce a scope

```
if constexpr(1 < 3 + 4)
{
    /* compile this */
}
else
{
    /* compile this */
}
```

The next big Thing [Andrei Alexandrescu, [Presentation](#), [Video](#)]



Über `constexpr`

if `constexpr` must not introduce a scope

```
if constexpr(1 < 3 + 4)
{
    /* compile this */
}
else
{
    /* compile this */
}
```

The next big Thing [Andrei Alexandrescu, [Presentation](#), [Video](#)]



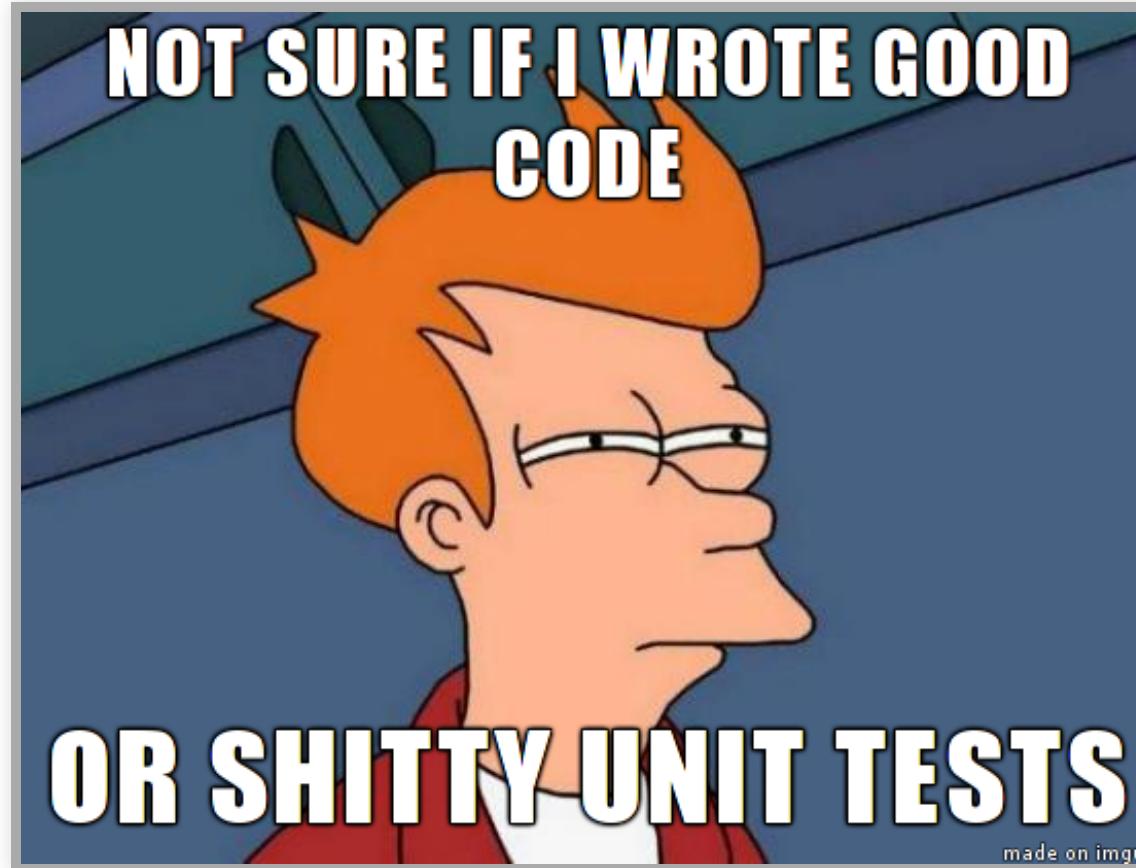
Über Kommentare

"A common fallacy is to assume authors of incomprehensible code will somehow be able to express themselves lucidly and clearly in comments."

[Kevlin Henney]



Über Testing



How to write more reliable code [Egor Bredikhin, [Presentation](#), [Video](#)]



Über Neuerfindungen



How to write more reliable code [Egor Bredikhin, Presentation, Video]



Über Neuerfindungen

Wieso

```
mySuperSort(v);
```

anstatt

How to write more reliable code [Egor Bredikhin, [Presentation](#), [Video](#)]



Über Neuerfindungen

Wieso

```
mySuperSort(v);
```

anstatt

```
std::sort(begin(v), end(v), std::greater<>());
```

How to write more reliable code [Egor Bredikhin, [Presentation](#), [Video](#)]



Über Neuerfindungen

Wieso

```
mySuperSort(v);
```

anstatt

```
std::sort(begin(v), end(v), std::greater<>());
```

*"Unless you are **an expert** in sorting algorithms and have plenty of time, this is more likely to be correct and to run faster than anything you write for a specific application. **You need a reason not to use the standard library rather than a reason to use it.**"*

[C++ Core Guidelines ©]

How to write more reliable code [Egor Bredikhin, [Presentation](#), [Video](#)]

Über Benamslungen



```
class ConditionChecker
{
public:
    virtual bool CheckCondition() const = 0;
}
```

Über Benamslungen



```
class ConditionChecker
{
public:
    virtual bool CheckCondition() const = 0;
}
```

Über Benamslungen



```
class ConditionChecker
{
public:
    virtual bool CheckCondition() const = 0;
}
```



Über Benamslungen



- Generell: Expliziter ist besser → aber am **richtigen Ort**:

```
class SelectionManager
{
public:

    class TriangulationSceneObjectProxyWrapper /* Stellvertr. */;

    TriangulationSceneObjectProxyWrapper&
    GetCurrentSelectedTriangulationObjectProxyWrapper();
};
```

Über Benamslungen



- Generell: Expliziter ist besser → aber am **richtigen Ort**:

```
class SelectionManager
{
public:

    class TriangulationSceneObjectProxyWrapper /* Stellvertr. */;

    TriangulationSceneObjectProxyWrapper&
    GetCurrentSelectedTriangulationObjectProxyWrapper();
};
```

Über Benamslungen



- Generell: Expliziter ist besser → aber am **richtigen Ort**:

```
class SelectionManager
{
public:

    class TriangulationSceneObjectProxyWrapper /* Stellvertr. */;

    TriangulationSceneObjectProxyWrapper&
    GetCurrentSelectedTriangulationObjectProxyWrapper();
};
```

Besser:

```
public:
    class TriObjectProxy { ... };
    TriObjectProxy& GetSelectedTriangulation();
}
```

Clean Coders Hate When You Use These Tricks [Kevlin Henney, [Video](#)]



Über Benamslungen

- Generell: Expliziter ist besser → aber am **richtigen Ort**:

```
class SelectionManager
{
public:

    class TriangulationSceneObjectProxyWrapper /* Stellvertr. */;

    TriangulationSceneObjectProxyWrapper&
    GetCurrentSelectedTriangulationObjectProxyWrapper();
};
```

Besser:

```
public:
    class TriObjectProxy { ... };
    TriObjectProxy& GetSelectedTriangulation();
}
```

Clean Coders Hate When You Use These Tricks [Kevlin Henney, [Video](#)]



Über Reflection & Tooling

Runtime-Reflection

Building a C++ Reflection System [Arvid Gerstman, [Presentation](#), [Video](#)]

```
Type t = user.GetType();
```



Über Reflection & Tooling

Runtime-Reflection

Building a C++ Reflection System [Arvid Gerstman, [Presentation](#), [Video](#)]

```
Type t = user.GetType();
```

- Suche im Abstract Syntax Tree (AST) mit einem Clang Tool (LLVM).



Über Reflection & Tooling

Runtime-Reflection

Building a C++ Reflection System [Arvid Gerstman, [Presentation](#), [Video](#)]

```
Type t = user.GetType();
```

- Suche im Abstract Syntax Tree (AST) mit einem Clang Tool (LLVM).
 - Annotierte Klassen:

```
struct __attribute__((annotate("reflect"))) User { ... }
```



Über Reflection & Tooling

Runtime-Reflection

Building a C++ Reflection System [Arvid Gerstman, [Presentation](#), [Video](#)]

```
Type t = user.GetType();
```

- Suche im Abstract Syntax Tree (AST) mit einem Clang Tool (LLVM).
 - Annotierte Klassen:

```
struct __attribute__((annotate("reflect"))) User { ... }
```

- 2-3 Tage Aufwand → Serialisierung



Über Reflection & Tooling

Runtime-Reflection

Building a C++ Reflection System [Arvid Gerstman, [Presentation](#), [Video](#)]

```
Type t = user.GetType();
```

- Suche im Abstract Syntax Tree (AST) mit einem Clang Tool (LLVM).
 - Annotierte Klassen:

```
struct __attribute__((annotate("reflect"))) User { ... }
```

- 2-3 Tage Aufwand → Serialisierung

Compile-Time Reflection

Compile-time programming and reflection in C++20... [Louis Dionne, [Video](#)]



Teil 1

Typen & Value-Kategorien



Teil 1



Typen & Value-Kategorien



Typen & Value-Kategorien





Typen & Value-Kategorien





History

Vor C++11: Binden eines temporären Objekts an eine **non-const** Referenz:

```
int& a = 0;
```

→ Compile Error ❌.

```
non-const lvalue reference to type 'int' cannot bind  
to a temporary of type 'int'
```



History

Vor C++11: Binden eines temporären Objekts an eine **non-const** Referenz:

```
int& a = 0;
```

→ Compile Error ❌.

```
non-const lvalue reference to type 'int' cannot bind  
to a temporary of type 'int'
```

Ab C++11: Neue Syntax für Referenzen auf temporäre Objekte:

```
int&& a = 0; // rvalue-Referenz auf `int`.
```



History

Vor C++11: Binden eines temporären Objekts an eine `non-const` Referenz:

```
int& a = 0;
```

→ Compile Error ❌.

```
non-const lvalue reference to type 'int' cannot bind  
to a temporary of type 'int'
```

Ab C++11: Neue Syntax für Referenzen auf temporäre Objekte:

```
int&& a = 0; // rvalue-Referenz auf `int`.
```

→ Effiziente Copy-Konstruktoren & Assignment-Operatoren etc.



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>      c; // Variable `c`: `std::vector<int>`  
  
int const * & const    d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```



Typ einer Variable

Jede Variabel besitzt ein Typ.

```
int                  a;
enum class C {A, B, C} b;
std::vector<int>    c; // Variable `c`: `std::vector<int>`  
  
int const * & const d;  
  
using FuncPointer = int (*)(float); // Typ Def.: Funktions-Pointer.  
FuncPointer pFunc;
```

- `decltype(a)` liefert den Typ der Variable `a`
- `template <typename T, ... > std::vector` ist kein Typ!
→ Template (siehe Teil 2)



Typ-Modifiers

- `const, volatile`
- `*, &`

gehören generell zum Typ und sind Modifikatoren.

Deshalb nicht zur Variabel:

```
const int *&d;
```



Wichtig, vor allem bei Templates → Teil 2.

```
int *e, f; // Gilt nicht als Ausrede (eher eine C++ Exception)
```



Typ einer Expression

Jede Expression `expr` besitzt zusätzlich zum **Typ** auch eine **Value-Kategorie**:

```
const int& b = 3;  
int c = b;  
//           ^----- Expr. Typ `const int`
```

Der Typ einer Expression ist **nie** eine Referenz weil Referenzen entfernt werden für jede weitere Analyse. **Siehe Regel [7.2.2#1]**



Typ einer Expression

Jede Expression `expr` besitzt zusätzlich zum **Typ** auch eine **Value-Kategorie**:

```
const int& b = 3;
int c = b;
//           ^----- Expr. Typ `const int`
```

Der Typ einer Expression ist **nie** eine Referenz weil Referenzen entfernt werden für jede weitere Analyse. **Siehe Regel [7.2.2#1]**

```
struct A{ A& operator+(const A&); };

A a, b;
a = a+b;
//     ^^^----- Expr. Typ ??
```



Typ einer Expression

Jede Expression `expr` besitzt zusätzlich zum **Typ** auch eine **Value-Kategorie**:

```
const int& b = 3;
int c = b;
//           ^----- Expr. Typ `const int`
```

Der Typ einer Expression ist **nie** eine Referenz weil Referenzen entfernt werden für jede weitere Analyse. **Siehe Regel [7.2.2#1]**

```
struct A{ A& operator+(const A&); };

A a, b;
a = a+b;
//     ^^^----- Expr. Typ ??
```

Beachte: Der Typ der Expression von `a+b` ist `A` !

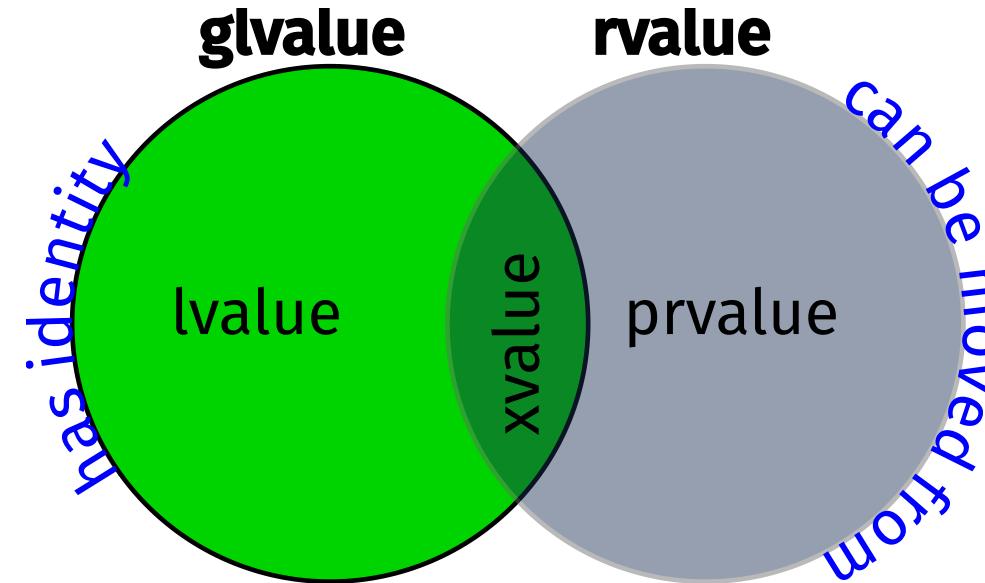


Value-Kategorien

Jede Expression `expr` besitzt zusätzlich zum Typ eine **Value-Category**:

```
const int& b = 3;  
int c = b;  
//           ^----- Expr. Typ `const int`
```

Kategorien C++14



- Für C++17 reicht das als Faustregel.

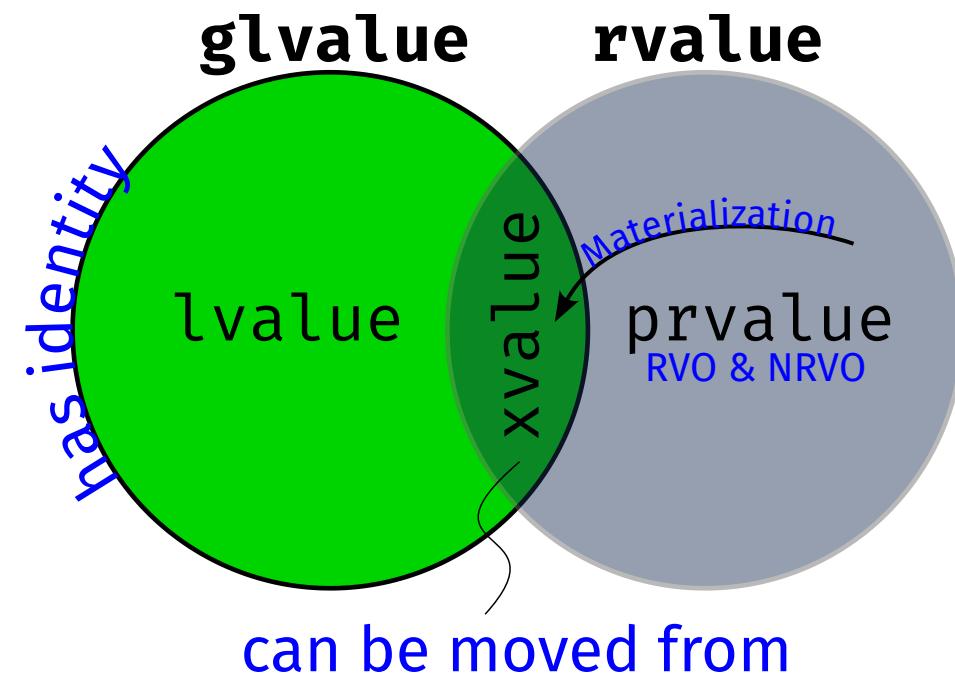


Value-Kategorien

Jede Expression `expr` besitzt zusätzlich zum Typ eine **Value-Category**:

```
const int& b = 3;  
int c = b;  
//           ^----- Expr. Typ `const int`
```

Kategorien C++17





Grundlegende Kategorien [basic.lval]

lvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **nicht wiederverwendet** werden kann. Alles von welchem man **die Adresse nehmen kann**.

```
int y;           // Variabel-Typ: 'int'  
int* p = &y;    // Variabel-Typ: 'int*'  
int& r = y;    // Variabel-Typ: 'int&'  
  
int&& rr = y;  // Variabel-Typ: 'int&&'
```

Expressions `y`, `p`, `r`, `rr` sind alles **lvalues**.



Grundlegende Kategorien [basic.lval]

lvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **nicht wiederverwendet** werden kann. Alles von welchem man **die Adresse nehmen kann**.

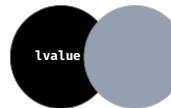
```
int y;           // Variabel-Typ: 'int'  
int* p = &y;    // Variabel-Typ: 'int*'  
int& r = y;    // Variabel-Typ: 'int&'  
  
int&& rr = y;  // Variabel-Typ: 'int&&'
```

Expressions `y`, `p`, `r`, `rr` sind alles **lvalues**.



Grundlegende Kategorien [basic.lval]

lvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **nicht wiederverwendet** werden kann. Alles von welchem man **die Adresse nehmen kann**.

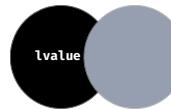
```
int y;           // Variabel-Typ: 'int'  
int* p = &y;    // Variabel-Typ: 'int*' (lvalue)  
int& r = y;    // Variabel-Typ: 'int&' (lvalue)  
  
int&& rr = y;   // Variabel-Typ: 'int&&' (lvalue)
```

Expressions `y`, `p`, `r`, `rr` sind alles **lvalues**.



Grundlegende Kategorien [basic.lval]

lvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **nicht wiederverwendet** werden kann. Alles von welchem man **die Adresse nehmen kann**.

```
int y;           // Variabel-Typ: 'int'  
int* p = &y;    // Variabel-Typ: 'int*'  
int& r = y;    // Variabel-Typ: 'int&'  
  
int&& rr = y;  // Variabel-Typ: 'int&&'
```

Expressions `y`, `p`, `r`, `rr` sind alles **lvalues**.



Grundlegende Kategorien [basic.lval]

lvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **nicht wiederverwendet** werden kann. Alles von welchem man **die Adresse nehmen kann**.

```
int y;           // Variabel-Typ: 'int'  
int* p = &y;    // Variabel-Typ: 'int*'  
int& r = y;    // Variabel-Typ: 'int&'  
  
int&& rr = y;  // Variabel-Typ: 'int&&'
```

Expressions `y`, `p`, `r`, `rr` sind alles **lvalues**.



Grundlegende Kategorien [basic.lval]

lvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **nicht wiederverwendet** werden kann. Alles von welchem man **die Adresse nehmen kann**.

```
int y;           // Variabel-Typ: 'int'  
int* p = &y;    // Variabel-Typ: 'int*'  
int& r = y;    // Variabel-Typ: 'int&'  
  
int&& rr = y;  // Variabel-Typ: 'int&&'
```

Expressions `y`, `p`, `r`, `rr` sind alles **lvalues**.

- Alle Variablen sind **immer lvalue**.



Grundlegende Kategorien [basic.lval]

rvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **wiederverwendet** werden darf. Meist weil am Ende seiner Laufzeit. Das beinhaltet **temporäre erstellte** Objekte (Materialisierung).



Grundlegende Kategorien [basic.lval]

rvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **wiederverwendet** werden darf. Meist weil am Ende seiner Laufzeit. Das beinhaltet **temporäre erstellte** Objekte (Materialisierung).

```
Banane make(){ return Banane{}; };
```



Grundlegende Kategorien [basic.lval]

rvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **wiederverwendet** werden darf. Meist weil am Ende seiner Laufzeit. Das beinhaltet **temporäre erstellte** Objekte (Materialisierung).

```
Banane make(){ return Banane{}; }
```

- Expression `Banane{}` → Typ `Banane` und **rvalue**



Grundlegende Kategorien [basic.lval]

rvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **wiederverwendet** werden darf. Meist weil am Ende seiner Laufzeit. Das beinhaltet **temporäre erstellte** Objekte (Materialisierung).

```
Banane make(){ return Banane{}; }
```

- Expression `Banane{}` → Typ `Banane` und **rvalue**

```
Banane& a = make();
```



Grundlegende Kategorien [basic.lval]

rvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **wiederverwendet** werden darf. Meist weil am Ende seiner Laufzeit. Das beinhaltet **temporäre erstellte** Objekte (Materialisierung).

```
Banane make(){ return Banane{}; }
```

- Expression `Banane{}` → Typ `Banane` und **rvalue**

```
Banane& a = make();
```

- Expression: `make()` : Typ `Banane` und **rvalue**



Grundlegende Kategorien [basic.lval]

rvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **wiederverwendet** werden darf. Meist weil am Ende seiner Laufzeit. Das beinhaltet **temporäre erstellte** Objekte (Materialisierung).

```
Banane make(){ return Banane{}; }
```

- Expression `Banane{}` → Typ `Banane` und **rvalue**

```
Banane& a = make();
```

- Expression: `make()` : Typ `Banane` und **rvalue**
- Variable `a` : Typ `Banane&`



Grundlegende Kategorien [basic.lval]

rvalue-Kategorie



Klassifiziert ein Objekt wessen Ressource **wiederverwendet** werden darf. Meist weil am Ende seiner Laufzeit. Das beinhaltet **temporäre erstellte** Objekte (Materialisierung).

```
Banane make(){ return Banane{}; }
```

- Expression `Banane{}` → Typ `Banane` und **rvalue**

```
Banane& a = make();
```

- Expression: `make()` : Typ `Banane` und **rvalue**
- Variable `a` : Typ `Banane&`
- Compile Error : `Banane&` bindet nicht an temporäre `Banane`.



Referenz-Typen

Syntax	Namen	Verhalten



Referenz-Typen

Syntax	Namen	Verhalten
Banane&	lvalue -Referenz	Bindet lvalues .



Referenz-Typen

Syntax	Namen	Verhalten
Banane&	lvalue -Referenz	Bindet lvalues .
const Banane&	const-lvalue -Referenz	Bindet lvalue und rvalue (weil const). [Ext. Life-Time]



Referenz-Typen

Syntax	Namen	Verhalten
Banane&	lvalue -Referenz	Bindet lvalues .
const Banane&	const-lvalue -Referenz	Bindet lvalue und rvalue (weil const). [Ext. Life-Time]
Banane&&	rvalue -Referenz	Bindet nur rvalues .



Referenz-Typen

Syntax	Namen	Verhalten
Banane&	lvalue -Referenz	Bindet lvalues .
const Banane&	const-lvalue -Referenz	Bindet lvalue und rvalue (weil const). [Ext. Life-Time]
Banane&&	rvalue -Referenz	Bindet nur rvalues .
T&& <small>[Template-Parameter T]</small> oder auto&&	forwarding -Referenz	Bindet an rvalue/lvalues . Um Value-Kategorie zu erhalten



Beispiel 1 [Live]

```
int&& a = 3; // 1. Value-Kategorie von `a`?  
int& b = a; // 2. Value-Kategorie von `b`?  
// 3. Kompiliert das?
```



Beispiel 1 [Live]

```
int&& a = 3; // 1. Value-Kategorie von `a`?  
int& b = a; // 2. Value-Kategorie von `b`?  
// 3. Kompiliert das?
```

1. Variable `a` hat Typ `int&&` [**rvalue**-Referenz auf `int`].
Expression `(a)` ist **lvalue** mit Typ: `int`.



Beispiel 1 [Live]

```
int&& a = 3; // 1. Value-Kategorie von `a`?  
int& b = a; // 2. Value-Kategorie von `b`?  
// 3. Kompiliert das?
```

1. Variable `a` hat Typ `int&&` [**rvalue**-Referenz auf `int`].
Expression `(a)` ist **lvalue** mit Typ: `int`.
2. Variable `b` hat Typ `int&` [**lvalue**-Referenz auf `int`].
Expression `(b)` ist **lvalue** mit Typ: `int`.



Beispiel 1 [Live]

```
int&& a = 3; // 1. Value-Kategorie von `a`?  
int& b = a; // 2. Value-Kategorie von `b`?  
// 3. Kompiliert das?
```

1. Variable `a` hat Typ `int&&` [**rvalue**-Referenz auf `int`].
Expression `(a)` ist **lvalue** mit Typ: `int`.
2. Variable `b` hat Typ `int&` [**lvalue**-Referenz auf `int`].
Expression `(b)` ist **lvalue** mit Typ: `int`.
3. Ja es kompiliert! ☺.



Value-Kategorie bei Rückgabewerten

[expr.call]

lvalue

rvalue

Jump to std::move ...



Value-Kategorie bei Rückgabewerten

[expr.call]

lvalue 

```
int& lvalue(); lvalue(); // Expression ist lvalue;
```

rvalue 

Jump to std::move ...



Value-Kategorie bei Rückgabewerten

[expr.call]

lvalue



```
int& lvalue(); lvalue(); // Expression ist lvalue;
```

rvalue



```
int&& xvalue(); xvalue(); // Expression ist xvalue;  
int prvalue(); prvalue(); // Expression ist prvalue;
```

Jump to std::move ...



Value-Kategorie bei Expression `f()`

[expr.call]

Rückgabetyp von `f()`

Expression `f()`

[Typ, Value-Kategorie]

Banane

→ [Banane, **prvalue**]

Banane&

→ [Banane, **lvalue**]

Banane&&

→ [Banane, **xvalue**]



Value-Kategorie bei Expression `f()`

[expr.call]

Rückgabetyp von <code>f()</code>	Expression <code>f()</code> [Typ, Value-Kategorie]
Banane	→ [Banane, prvalue]
Banane&	→ [Banane, lvalue]
Banane&&	→ [Banane, xvalue]

Inverses Mapping macht `decltype(expr)`:

decltype(expr)	Expression <code>expr</code>
Banane	← [Banane, prvalue]
Banane&	← [Banane, lvalue]
Banane&&	← [Banane, xvalue]

Bindungsverhalten



`int& lvalue();`

`int&& xvalue();`

`int prvalue();`

lvalue-Referenz

rvalue-Referenz

Live

Bindungsverhalten



int& lvalue();

int&& xvalue();

int prvalue();

lvalue-Referenz

int& r = lvalue();	// ☺:	lvalue-Ref. bindet an lvalue.
int& r = prvalue();	// ❌:	bindet nicht an prvalue.
int& r = xvalue();	// ❌:	bindet nicht an xvalue.

rvalue-Referenz



Bindungsverhalten

int& lvalue();

int&& xvalue();

int prvalue();

lvalue-Referenz

int& r = lvalue();	// ☺:	lvalue-Ref. bindet an lvalue.
int& r = prvalue();	// ❌:	bindet nicht an prvalue.
int& r = xvalue();	// ❌:	bindet nicht an xvalue.

const int& r = prvalue(); // ☺:	bindet an prvalue.
const int& r = xvalue(); // ☺:	bindet an xvalue.

rvalue-Referenz



Bindungsverhalten

int& lvalue();

int&& xvalue();

int prvalue();

lvalue-Referenz

int& r = lvalue();	// ☺:	lvalue-Ref. bindet an lvalue.
int& r = prvalue();	// ❌:	bindet nicht an prvalue.
int& r = xvalue();	// ❌:	bindet nicht an xvalue.

const int& r = prvalue(); // ☺:	bindet an prvalue.
const int& r = xvalue(); // ☺:	bindet an xvalue.

rvalue-Referenz

int&& rr = lvalue();	// ❌:	rvalue-Ref. bindet nicht an lvalue.
int&& rr = prvalue();	// ☺:	bindet an prvalue.
int&& rr = xvalue();	// ☺:	bindet an xvalue.



Bindungsverhalten

`int& lvalue();`

`int&& xvalue();`

`int prvalue();`

lvalue-Referenz

<code>int& r = lvalue();</code>	// ☺:	lvalue-Ref. bindet an lvalue.
<code>int& r = prvalue();</code>	// ❌:	bindet nicht an prvalue.
<code>int& r = xvalue();</code>	// ❌:	bindet nicht an xvalue.

<code>const int& r = prvalue();</code>	// ☺:	bindet an prvalue.
<code>const int& r = xvalue();</code>	// ☺:	bindet an xvalue.

rvalue-Referenz

<code>int&& rr = lvalue();</code>	// ❌:	rvalue-Ref. bindet nicht an lvalue.
<code>int&& rr = prvalue();</code>	// ☺:	bindet an prvalue.
<code>int&& rr = xvalue();</code>	// ☺:	bindet an xvalue.

- Bindet nur an **rvalues!**



Wieso rvalue-Referenzen && ?

```
class Banane
{
public:
    Banane(const Banane& rOther);           // Copy-Konstruktor
    Banane(Banane&& rrOther);               // Move-Konstruktor;

    Banane& operator=(const Banane& rOther); // Assign-Operator
    Banane& operator=(Banane&& rrOther);    // Move-Assign-Operator;
};
```



Wieso rvalue-Referenzen **&&** ?

```
class Banane
{
public:
    Banane(const Banane& rOther);           // Copy-Konstruktor
    Banane(Banane&& rrOther);               // Move-Konstruktor;

    Banane& operator=(const Banane& rOther); // Assign-Operator
    Banane& operator=(Banane&& rrOther);    // Move-Assign-Operator;
};
```

- Effizienter "Copy"-Konstruktor möglich → **Move-Konstruktor**.



Wieso rvalue-Referenzen && ?

```
class Banane
{
public:
    Banane(const Banane& rOther);           // Copy-Konstruktor
    Banane(Banane&& rrOther);               // Move-Konstruktor;

    Banane& operator=(const Banane& rOther); // Assign-Operator
    Banane& operator=(Banane&& rrOther);     // Move-Assign-Operator;
};
```

- Effizienter "Copy"-Konstruktor möglich → **Move-Konstruktor**.
- Effizienter "Assign"-Operator möglich → **Move-Assign-Operator**.



Wieso rvalue-Referenzen && ?

```
class Banane
{
public:
    Banane(const Banane& rOther);           // Copy-Konstruktor
    Banane(Banane&& rrOther);               // Move-Konstruktor;

    Banane& operator=(const Banane& rOther); // Assign-Operator
    Banane& operator=(Banane&& rrOther);     // Move-Assign-Operator;
};
```

```
Banane create();
Banane a = create();
```



Wieso rvalue-Referenzen && ?

```
class Banane
{
public:
    Banane(const Banane& rOther);           // Copy-Konstruktor
    Banane(Banane&& rrOther);               // Move-Konstruktor;

    Banane& operator=(const Banane& rOther); // Assign-Operator
    Banane& operator=(Banane&& rrOther);    // Move-Assign-Operator;
};
```

```
Banane create();
Banane a = create();
```

- `create()` liefert **rvalue** mit Expr. Typ `Banane`.



Wieso rvalue-Referenzen && ?

```
class Banane
{
public:
    Banane(const Banane& rOther);           // Copy-Konstruktor
    Banane(Banane&& rrOther);               // Move-Konstruktor;

    Banane& operator=(const Banane& rOther); // Assign-Operator
    Banane& operator=(Banane&& rrOther);     // Move-Assign-Operator;
};
```

```
Banane create();
Banane a = create();
```

- `create()` liefert **prvalue** mit Expr. Typ `Banane`.
- Der Move-Konstruktor (falls es ihn gibt) wird dem Copy-Konstruktor bevorzugt, weil `Banane&&` besser matched.



Debugging Value-Kategorien

Benutze `decltype((expr))`:



Debugging Value-Kategorien

Benutze `decltype((expr))`:

- `T` falls `expr` ein **prvalue** ist.



Debugging Value-Kategorien

Benutze `decltype((expr))`:

- `T` falls `expr` ein **prvalue** ist.
- `T&` falls `expr` ein **lvalue** ist.



Debugging Value-Kategorien

Benutze `decltype((expr))`:

- `T` falls `expr` ein **prvalue** ist.
- `T&` falls `expr` ein **lvalue** ist.
- `T&&` falls `expr` ein **xvalue** ist.



Debugging Value-Kategorien

Benutze `decltype((expr))`:

- `T` falls `expr` ein **prvalue** ist.
- `T&` falls `expr` ein **lvalue** ist.
- `T&&` falls `expr` ein **xvalue** ist.

Wieso `(expr)`: Weil explizit so beschrieben in [dcl.type decltype]



Debugging Value-Kategorien

Benutze `decltype((expr))`:

- `T` falls `expr` ein **prvalue** ist.
- `T&` falls `expr` ein **lvalue** ist.
- `T&&` falls `expr` ein **xvalue** ist.

Wieso `(expr)`: Weil explizit so beschrieben in [dcl.type decltype]

```
int a;  
decltype((a))::GUGUS // Kompilierfehler ☹
```



Debugging Value-Kategorien

Benutze `decltype((expr))`:

- `T` falls `expr` ein **prvalue** ist.
- `T&` falls `expr` ein **lvalue** ist.
- `T&&` falls `expr` ein **xvalue** ist.

Wieso `(expr)`: Weil explizit so beschrieben in [dcl.type decltype]

```
int a;  
decltype((a))::GUGUS // Kompilierfehler ❌
```

Ausgabe:

```
error: 'decltype((a))' (aka 'int &') is not a class,  
namespace, or enumeration
```



Debugging Value-Kategorien

Benutze `decltype((expr))`:

- `T` falls `expr` ein **prvalue** ist.
- `T&` falls `expr` ein **lvalue** ist.
- `T&&` falls `expr` ein **xvalue** ist.

Wieso `(expr)`: Weil explizit so beschrieben in [dcl.type decltype]

```
int a;  
decltype((a))::GUGUS // Kompilierfehler ❌
```

Ausgabe:

```
error: 'decltype((a))' (aka 'int &') is not a class,  
namespace, or enumeration
```

- → `a` ist **lvalue**.



Was macht std :: move(...)

Transformiert die Value-Kategorie einer Expression von **lvalue** nach **xvalue**.

```
struct A{}; A a; // 'a' ist lvalue.  
std::move(a); // ... eine xvalue-Expression.
```



Was macht std::move(...)

Transformiert die Value-Kategorie einer Expression von **lvalue** nach **xvalue**.

```
struct A{}; A a; // 'a' ist lvalue.  
std::move(a); // ... eine xvalue-Expression.
```

Macht folgendes:

```
A&& std::move( ... ) { return static_cast<A&&>(a); }
```

Somit ist `std::move(a)` ein **xvalue** vom Expr. Typ: `A`,
welches zu Aufruf



Was macht std :: move(...)

Transformiert die Value-Kategorie einer Expression von **lvalue** nach **xvalue**.

```
struct A{}; A a; // 'a' ist lvalue.  
std::move(a); // ... eine xvalue-Expression.
```

Macht folgendes: → siehe [xvalue\(\)](#).

```
A&& std::move( ... ) { return static_cast<A&&>(a); }
```

Somit ist `std :: move(a)` ein **xvalue** vom Expr. Typ: `A`,
welches zu Aufruf



Was macht std :: move(...)

Transformiert die Value-Kategorie einer Expression von **lvalue** nach **xvalue**.

```
struct A{}; A a; // 'a' ist lvalue.  
std::move(a); // ... eine xvalue-Expression.
```

Macht folgendes: → siehe [xvalue\(\)](#).

```
A&& std::move( ... ) { return static_cast<A&&>(a); }
```

Somit ist `std :: move(a)` ein **xvalue** vom Expr. Typ: `A`,
welches zu Aufruf

```
A(A&& a); // Move-Konstruktor.  
A& operator=(A&& a); // Move-Zuweisungsoperator.
```

führen **kann**



Was macht std :: move(...)

Transformiert die Value-Kategorie einer Expression von **lvalue** nach **xvalue**.

```
struct A{}; A a; // 'a' ist lvalue.  
std::move(a); // ... eine xvalue-Expression.
```

Macht folgendes: → siehe [xvalue\(\)](#).

```
A&& std::move( ... ) { return static_cast<A&&>(a); }
```

Somit ist `std :: move(a)` ein **xvalue** vom Expr. Typ: `A`,
welches zu Aufruf

```
A(A&& a); // Move-Konstruktor.  
A& operator=(A&& a); // Move-Zuweisungsoperator.
```

führen **kann** was dann **Move** heisst:



Was macht std :: move(...)

Transformiert die Value-Kategorie einer Expression von **lvalue** nach **xvalue**.

```
struct A{}; A a; // 'a' ist lvalue.  
std::move(a); // ... eine xvalue-Expression.
```

Macht folgendes: → siehe [xvalue\(\)](#).

```
A&& std::move( ... ) { return static_cast<A&&>(a); }
```

Somit ist `std :: move(a)` ein **xvalue** vom Expr. Typ: `A`,
welches zu Aufruf

```
A(A&& a); // Move-Konstruktor.  
A& operator=(A&& a); // Move-Zuweisungsoperator.
```

führen **kann** was dann **Move** heisst:

```
A b = std::move(a); // → Move-Konstruktor
```



Beispiel 2 - Return bei Referenz

```
class MpAlgo
{
public:
    Triangulation& GetOutput() { return m_Tri; };
};

MpAlgo algo;
Triangulation r1 = algo.GetOutput();           // 1. Typ/Val.Kat?
Triangulation r2 = std::move(algo.GetOutput()); // 2. Typ/Val.Kat?
                                                // 3. m_Tri?
```



Beispiel 2 - Return bei Referenz

```
class MpAlgo
{
public:
    Triangulation& GetOutput() { return m_Tri; };
};

MpAlgo algo;
Triangulation r1 = algo.GetOutput();           // 1. Typ/Val.Kat?
Triangulation r2 = std::move(algo.GetOutput()); // 2. Typ/Val.Kat?
                                                // 3. m_Tri?
```



Beispiel 2 - Return bei Referenz

```
class MpAlgo
{
public:
    Triangulation& GetOutput() { return m_Tri; }
};

MpAlgo algo;
Triangulation r1 = algo.GetOutput();           // 1. Typ/Val.Kat?
Triangulation r2 = std::move(algo.GetOutput()); // 2. Typ/Val.Kat?
                                                // 3. m_Tri?
```

1. Expr. Typ **Triangulation** und **lvalue** → Copy-Konstruktor.



Beispiel 2 - Return bei Referenz

```
class MpAlgo
{
public:
    Triangulation& GetOutput() { return m_Tri; };
};

MpAlgo algo;
Triangulation r1 = algo.GetOutput();           // 1. Typ/Val.Kat?
Triangulation r2 = std::move(algo.GetOutput()); // 2. Typ/Val.Kat?
                                                // 3. m_Tri?
```

1. Expr. Typ **Triangulation** und **lvalue** → Copy-Konstruktor.



Beispiel 2 - Return bei Referenz

```
class MpAlgo
{
public:
    Triangulation& GetOutput() { return m_Tri; }
};

MpAlgo algo;
Triangulation r1 = algo.GetOutput();           // 1. Typ/Val.Kat?
Triangulation r2 = std::move(algo.GetOutput()); // 2. Typ/Val.Kat?
                                                // 3. m_Tri?
```

1. Expr. Typ `Triangulation` und **lvalue** → Copy-Konstruktor.
2. Expr. Typ `Triangulation` und **xvalue** → Move-Konstruktor.



Beispiel 2 - Return bei Referenz

```
class MpAlgo
{
public:
    Triangulation& GetOutput() { return m_Tri; };
};

MpAlgo algo;
Triangulation r1 = algo.GetOutput();           // 1. Typ/Val.Kat?
Triangulation r2 = std::move(algo.GetOutput()); // 2. Typ/Val.Kat?
                                                // 3. m_Tri?
```

1. Expr. Typ `Triangulation` und **lvalue** → Copy-Konstruktor.
2. Expr. Typ `Triangulation` und **xvalue** → Move-Konstruktor.
3. `m_Tri` ist in einem undefinierten Zustand (nach Standard)



Beispiel 2 - Return bei Referenz

```
class MpAlgo
{
public:
    Triangulation& GetOutput() { return m_Tri; };
};

MpAlgo algo;
Triangulation r1 = algo.GetOutput();           // 1. Typ/Val.Kat?
Triangulation r2 = std::move(algo.GetOutput()); // 2. Typ/Val.Kat?
                                                // 3. m_Tri?
```

1. Expr. Typ `Triangulation` und **lvalue** → Copy-Konstruktor.
2. Expr. Typ `Triangulation` und **xvalue** → Move-Konstruktor.
3. `m_Tri` ist in einem undefinierten Zustand (nach Standard)
 - Aufpassen bei Wiederverwendung von `algo`.



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```

1. `createIds()` : `Array<int>` und **prvalue** → Move-Konstruktor falls vorhanden sonst Copy-Konstruktor.



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```

1. `createIds()` : `Array<int>` und **prvalue** → Move-Konstruktor falls vorhanden sonst Copy-Konstruktor.
 - kein `std::move(ids)`! → Compiler kann nicht optimieren.



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```

1. `createIds()` : `Array<int>` und **prvalue** → Move-Konstruktor falls vorhanden sonst Copy-Konstruktor.
 - kein `std :: move(ids)`! → Compiler kann nicht optimieren.
 - **Named Return Value Optimization (NRVO)** : [11.9.5] [cppref] [Slides]



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```

1. `createIds()` : `Array<int>` und **prvalue** → Move-Konstruktor falls vorhanden sonst Copy-Konstruktor.
 - kein `std :: move(ids)`! → Compiler kann nicht optimieren.
 - **Named Return Value Optimization (NRVO)** : [11.9.5] [cppref] [Slides]
 - NRVO: **Named** Objekt einer **prvalue** Return Expression.



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```

1. `createIds()` : `Array<int>` und **prvalue** → Move-Konstruktor falls vorhanden sonst Copy-Konstruktor.
 - kein `std :: move(ids)`! → Compiler kann nicht optimieren.
 - **Named Return Value Optimization (NRVO)** : [11.9.5] [cppref] [Slides]
 - NRVO: **Named** Objekt einer **prvalue** Return Expression.
 - Für Compiler immer noch optional in C++17/20.



Beispiel 3 - Return bei Value

```
Array<int> createIds() {  
    Array<int> ids;  
    for(int i=0; i<42; ++i){  
        ids.AddTail(i);  
    }  
    return ids;  
};  
Array<int> ids = createIds(); // 1) Typ / Value-Kategorie
```

1. `createIds()` : `Array<int>` und **prvalue** → Move-Konstruktor falls vorhanden sonst Copy-Konstruktor.
 - kein `std :: move(ids)`! → Compiler kann nicht optimieren.
 - **Named Return Value Optimization (NRVO)** : [11.9.5] [cppref] [Slides]
 - NRVO: **Named** Objekt einer **prvalue** Return Expression.
 - Für Compiler immer noch optional in C++17/20.

Return Value Optimization (RVO) ist verpflichtend seit C++17



Beispiel 4 - Return bei Value & RVO

```
Array<int> createIds() {  
    return {1, 2, 3, 4};  
};  
Array<int> ids = createIds(); // 1.
```



Beispiel 4 - Return bei Value & RVO

```
Array<int> createIds() {  
    return {1, 2, 3, 4};  
}  
Array<int> ids = createIds(); // 1.
```

1. Return Value Optimization (RVO) : [11.9.5] [cppref] [Slides]

- RVO: **Unnamed** Objekt einer **prvalue** Return Expression



Beispiel 4 - Return bei Value & RVO

```
Array<int> createIds() {  
    return {1, 2, 3, 4};  
}  
Array<int> ids = createIds(); // 1.
```

1. Return Value Optimization (RVO) : [11.9.5] [cppref] [Slides]

- RVO: **Unnamed** Objekt einer **prvalue** Return Expression
- Für Compiler **verpflichtend** in C++17/20.



Questions



To be continued

Teil 2: Template Refresher, Ref. Collapsing, `std :: forward`, Template Argument Deduktion, C++17, Code Simplify etc. (vielleicht auch Teil 3 ?)



Referenzen

- [C++2x Standard]
- [C++ Meeting 2018 Slides]
- [Value Categories in C++17]
- [Effective Modern C++, 2015]
 - Kapitel 6. Rvalue-Referenzen, Move-Semantik und Perfect-Forwarding.
- [C++ Templates: The Complete Guide]
 - Appendix B: Value Categories. Empfehlenswert. IMO: Diese Slides fassen das aber besser zusammen.



Bemerkungen zu Teil 1



Bemerkungen zu Teil 1

- Für die Anwendung reicht es aus zwischen **rvalue** und **lvalue** zu unterscheiden:



Bemerkungen zu Teil 1

- Für die Anwendung reicht es aus zwischen **rvalue** und **lvalue** zu unterscheiden:
 - `std :: move(banane)` ergibt eine Expression mit Value-Kategorie **xvalue**:
→ "kann gemoved werden"
d.h. bei Konstruktion oder Zuweisung matcht **Move-Konstruktor/Move-Assigment** besser. Fallback ist immer Copy-Konstruktor/Copy-Assigment.



Bemerkungen zu Teil 1

- Für die Anwendung reicht es aus zwischen **rvalue** und **lvalue** zu unterscheiden:
 - `std :: move(banane)` ergibt eine Expression mit Value-Kategorie **xvalue**:
→ "kann gemoved werden"
d.h. bei Konstruktion oder Zuweisung matcht **Move-Konstruktor/Move-Assigment** besser. Fallback ist immer Copy-Konstruktor/Copy-Assigment.
- Typen von Expressions sind **nie** Referenzen: [1], [2], [2], [3]



Bemerkungen zu Teil 1

- Für die Anwendung reicht es aus zwischen **rvalue** und **lvalue** zu unterscheiden:
 - `std :: move(banane)` ergibt eine Expression mit Value-Kategorie **xvalue**:
→ "kann gemoved werden"
d.h. bei Konstruktion oder Zuweisung matcht **Move-Konstruktor/Move-Assigment** besser. Fallback ist immer Copy-Konstruktor/Copy-Assigment.
- Typen von Expressions sind **nie** Referenzen: [1], [2], [2], [3]
- Nur [[Venn-Diagram](#)] für C++17 wichtig!



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {
    A(const A&) = delete;
    A(const A&&) = delete;
};

A nrvo() { A a; return a; };
A rvo() { return A{}; };
A rvo2() { return rvo(); };

A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {  
    A(const A&) = delete;  
    A(const A&&) = delete;  
};  
  
A nrvo() { A a; return a; };  
A rvo() { return A{}; };  
A rvo2() { return rvo(); };  
  
A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.  
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.  
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {  
    A(const A&) = delete;  
    A(const A&&) = delete;  
};  
  
A nrvo() { A a; return a; };  
A rvo() { return A{}; };  
A rvo2() { return rvo(); };  
  
A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.  
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.  
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {  
    A(const A&) = delete;  
    A(const A&&) = delete;  
};  
  
A nrvo() { A a; return a; };  
A rvo() { return A{}; };  
A rvo2() { return rvo(); };  
  
A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.  
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.  
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {  
    A(const A&) = delete;  
    A(const A&&) = delete;  
};  
  
A nrvo() { A a; return a; };  
A rvo() { return A{}; };  
A rvo2() { return rvo(); };  
  
A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.  
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.  
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {  
    A(const A&) = delete;  
    A(const A&&) = delete;  
};  
  
A nrvo() { A a; return a; };  
A rvo() { return A{}; };  
A rvo2() { return rvo(); };  
  
A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.  
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.  
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {  
    A(const A&) = delete;  
    A(const A&&) = delete;  
};  
  
A nrvo() { A a; return a; };  
A rvo() { return A{}; };  
A rvo2() { return rvo(); };  
  
A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.  
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.  
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Recap 1 : [Named]-Return-Value-Optimization [Live]

Es geht um prvalues in einer Return-Expression:

```
struct A {  
    A(const A&) = delete;  
    A(const A&&) = delete;  
};  
  
A nrvo() { A a; return a; };  
A rvo() { return A{}; };  
A rvo2() { return rvo(); };  
  
A a = nrvo(); // NRVO optional , kompiliert nicht, weil `=delete`.  
A b = rvo(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.  
A c = rvo2(); // RVO verpflichtend, kompiliert auch ohne Move/Copy.
```

- C++17 garantiert copy-elision für **RVO** Fälle wie `rvo()`, `rvo2()`.



Wann std :: move(...) ?



Wann std :: move(...) ?

- **Moven** heisst Move-Konstruktor/Assignment aufrufen.



Wann std :: move(...) ?

- **Moven** heisst Move-Konstruktor/Assignment aufrufen.
- Objekte die **billig zu moven** sind



Wann std :: move(...) ?

- **Moven** heisst Move-Konstruktor/Assignment aufrufen.
- Objekte die **billig zu moven** sind
 - **Value-Semantik:** *Return/Call-By-Value*



Wann std :: move(...) ?

- **Moven** heisst Move-Konstruktor/Assignment aufrufen.
- Objekte die **billig zu moven** sind
 - **Value-Semantik:** Return/Call-By-Value
 - **Generell nie** return std :: move(ret) benutzen:

```
A create() { A ret; ... return ret; }; // NRVO.
```



Wann std :: move(...) ?

- **Moven** heisst Move-Konstruktor/Assignment aufrufen.
- Objekte die **billig zu moven** sind
 - **Value-Semantik:** Return/Call-By-Value
 - **Generell nie** return std :: move(ret) benutzen:

```
A create() { A ret; ... return ret; }; // NRVO.
```

- Objekte die **teuer zu moven** sind, z.B. Triangulation:



Wann std :: move(...) ?

- **Moven** heisst Move-Konstruktor/Assignment aufrufen.
- Objekte die **billig zu moven** sind
 - **Value-Semantik:** *Return/Call-By-Value*
 - **Generell nie** `return std::move(ret)` benutzen:

```
A create() { A ret; ... return ret; }; // NRVO.
```

- Objekte die **teuer zu moven** sind, z.B. `Triangulation`:
 - **Eher Referenz-Semantik:** *Return/Call-By-Reference*



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression a ist **lvalue** und Typ Banane.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression a ist **lvalue** und Typ Banane.
- 2.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
|                                     // 3. Typ/Val.Kat von `std::move(a)`?  
                                      // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
- 3.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
|                                     // 3. Typ/Val.Kat von `std::move(a)`?  
                                      // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.
- 4.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.
4. `a` ist `removed`, `a` ist in **unspecified aber validen Zustand**.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.
4. `a` ist **removed**, `a` ist in **unspecified aber validen Zustand**.
Man sollte nichts anderes annehmen, auch wenn kopiert wurde in 2.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.
4. `a` ist **removed**, `a` ist in **unspecified aber validen Zustand**.
Man sollte nichts anderes annehmen, auch wenn kopiert wurde in 2.
- 5.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.
4. `a` ist gemoved, `a` ist in **unspezifiziertem aber validen Zustand**.
Man sollte nichts anderes annehmen, auch wenn kopiert wurde in 2.
5. `shake(const Banane&)`.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.
4. `a` ist **removed**, `a` ist in **unspezifiziertem aber validen Zustand**.
Man sollte nichts anderes annehmen, auch wenn kopiert wurde in 2.
5. `shake(const Banane&)`.
- 6.



Recap 2 [Live]

```
struct Banane{};  
void shake(const Banane& rB);  
void shake(Banane&& rrB);  
  
Banane a;  
a;                                // 1. Typ/Val.Kat von Expression `a` ?  
Banane b = std::move(a);           // 2. Rückgabe-Typ von `std::move(a)` ?  
                                  // 3. Typ/Val.Kat von `std::move(a)`?  
                                  // 4. Was ist der Zustand von `a` ?  
shake(b);                          // 5. Welche Funktion wird aufgerufen?  
shake(std::move(b));              // 6. Welche Funktion wird aufgerufen?
```

1. Expression `a` ist **lvalue** und Typ `Banane`.
2. Rückgabetyp von `std :: move(b)` ist `Banane&&`.
3. Expression `std :: move(b)` ist **xvalue** und Typ `Banane`.
4. `a` ist **removed**, `a` ist in **unspezifiziertem aber validen Zustand**.
Man sollte nichts anderes annehmen, auch wenn kopiert wurde in 2.
5. `shake(const Banane&)`.
6. `shake(Banane&&)` matcht das **xvalue** von `std :: move(b)` besser.



Recap 3 [Live]

```
struct Banane{};  
void foo(const Banane& rB);  
void foo(Banane&& rrB);  
  
Banane a;  
Banane&& b = std::move(a); // 1. Was ist der Zustand von `a` ?  
foo(b); // 2. Welche Funktion wird aufgerufen?  
foo(std::move(b)); // 3. Welche Funktion wird aufgerufen?
```



Recap 3 [Live]

```
struct Banane{};  
void foo(const Banane& rB);  
void foo(Banane&& rrB);  
  
Banane a;  
Banane&& b = std::move(a); // 1. Was ist der Zustand von `a` ?  
foo(b); // 2. Welche Funktion wird aufgerufen?  
foo(std::move(b)); // 3. Welche Funktion wird aufgerufen?
```

1.



Recap 3 [Live]

```
struct Banane{};  
void foo(const Banane& rB);  
void foo(Banane&& rrB);  
  
Banane a;  
Banane&& b = std::move(a); // 1. Was ist der Zustand von `a` ?  
foo(b); // 2. Welche Funktion wird aufgerufen?  
foo(std::move(b)); // 3. Welche Funktion wird aufgerufen?
```

1. Der Speicher von a hat sich nicht geändert!



Recap 3 [Live]

```
struct Banane{};  
void foo(const Banane& rB);  
void foo(Banane&& rrB);  
  
Banane a;  
Banane&& b = std::move(a); // 1. Was ist der Zustand von `a` ?  
foo(b); // 2. Welche Funktion wird aufgerufen?  
foo(std::move(b)); // 3. Welche Funktion wird aufgerufen?
```

1. Der Speicher von a hat sich nicht geändert!
- 2.



Recap 3 [Live]

```
struct Banane{};  
void foo(const Banane& rB);  
void foo(Banane&& rrB);  
  
Banane a;  
Banane&& b = std::move(a); // 1. Was ist der Zustand von `a` ?  
foo(b); // 2. Welche Funktion wird aufgerufen?  
foo(std::move(b)); // 3. Welche Funktion wird aufgerufen?
```

1. Der Speicher von a hat sich nicht geändert!
2. foo(const Banane&). Achtung: b is lvalue!!



Recap 3 [Live]

```
struct Banane{};  
void foo(const Banane& rB);  
void foo(Banane&& rrB);  
  
Banane a;  
Banane&& b = std::move(a); // 1. Was ist der Zustand von `a` ?  
foo(b); // 2. Welche Funktion wird aufgerufen?  
foo(std::move(b)); // 3. Welche Funktion wird aufgerufen?
```

1. Der Speicher von `a` hat sich nicht geändert!
2. `foo(const Banane&)`. Achtung: `b` is **lvalue!!**
- 3.



Recap 3 [Live]

```
struct Banane{};  
void foo(const Banane& rB);  
void foo(Banane&& rrB);  
  
Banane a;  
Banane&& b = std::move(a); // 1. Was ist der Zustand von `a` ?  
foo(b); // 2. Welche Funktion wird aufgerufen?  
foo(std::move(b)); // 3. Welche Funktion wird aufgerufen?
```

1. Der Speicher von `a` hat sich nicht geändert!
2. `foo(const Banane&)`. Achtung: `b` is **lvalue!!**
3. `foo(Banane&&)` matcht das **xvalue** von `std :: move(b)` besser.



Recap 4 - Move-Semantik schreiben

```
struct A
{
    A(const A& rOther){
        m_v = rOther.m_v;
    }
    A(A&& rrOther){
        m_v = rrOther.m_v; // 1. Wieso ist das hier suboptimal?
    }
    std::vector<int> m_v;
};
```



Recap 4 - Move-Semantik schreiben

```
struct A
{
    A(const A& rOther){
        m_v = rOther.m_v;
    }
    A(A&& rrOther){
        m_v = rrOther.m_v; // 1. Wieso ist das hier suboptimal?
    }
    std::vector<int> m_v;
};
```

1.



Recap 4 - Move-Semantik schreiben

```
struct A
{
    A(const A& rOther){
        m_v = rOther.m_v;
    }
    A(A&& rrOther){
        m_v = rrOther.m_v; // 1. Wieso ist das hier suboptimal?
    }
    std::vector<int> m_v;
};
```

1. Expression `rrOther` ist **lvalue**, d.h. hier wird kopiert. Die Variable `rrOther` ist aber eine Referenz auf ein temporäres A [**rvalue**-Reference].



Recap 4 - Move-Semantik schreiben

```
struct A
{
    A(const A& rOther){
        m_v = rOther.m_v;
    }
    A(A&& rrOther){
        m_v = rrOther.m_v; // 1. Wieso ist das hier suboptimal?
    }
    std::vector<int> m_v;
};
```

1. Expression `rrOther` ist **lvalue**, d.h. hier wird kopiert. Die Variable `rrOther` ist aber eine Referenz auf ein temporäres A [**rvalue**-Reference].
Richtig wäre: `std::move(rrOther.m_v)` damit das retournierte **xvalue** (`std::vector<int>&&`) den **Move-Assign-Operator** matcht.

Recap 5 - Referenzen zurück geben. 💣



```
struct A{};  
  
A&      get() { return A{}; }           // No!!!  
A&      get() { A a; return a; }         // No!!!  
const A& get() { return A{}; }         // No!!!  
A&&    get() { return A{}; }           // No!!!  
A&&    get() { A a; return std::move(a); } // No!!!  
  
auto& get() { return A{}; }           // No!!!  
auto&& get() { return A{}; }           // No!!!
```

Recap 5 - Referenzen zurück geben. 💣



```
struct A{};  
  
A&      get() { return A{}; }           // No!!!  
A&      get() { A a; return a; }        // No!!!  
const A& get() { return A{}; }        // No!!!  
A&&     get() { return A{}; }           // No!!!  
A&&     get() { A a; return std::move(a); } // No!!!  
  
auto& get() { return A{}; }           // No!!!  
auto&& get() { return A{}; }           // No!!!
```

- **Wichtig:** Referenzen zurückgeben auf **lokale temporäre** Objekte ist **immer quatsch!** Compiler sollte warnen!

Recap 5 - Referenzen zurück geben. 💣



```
struct A{};  
  
A&      get() { return A{}; }           // No!!!  
A&      get() { A a; return a; }        // No!!!  
const A& get() { return A{}; }        // No!!!  
A&&     get() { return A{}; }           // No!!!  
A&&     get() { A a; return std::move(a); } // No!!!  
  
auto& get() { return A{}; }           // No!!!  
auto&& get() { return A{}; }          // No!!!
```

- **Wichtig:** Referenzen zurückgeben auf **lokale temporäre** Objekte ist **immer quatsch!** Compiler sollte warnen!

Die **einzigsten richtigen Signaturen sind:**

```
A get() { return A{}; }  
A get() { return A{}; }  
auto get() { return A{}; } // kein auto& oder auto&& !!
```



Recap 6 - Aus Best Practice Chat [Live]

```
void SetData(A& a);
A a;
SetData(std::move(a)); // Darf nicht kompilieren! ❌:
```

- Eine **lvalue**-Referenz bindet nicht an **rvalues**.
- MSVC erlaubt das trotzdem [Live] obwohl **nicht-standard konform**. → 🤯



C++17 Features - Code Simplify

Init-If

```
if (auto val = GetValue(); condition(val))  
{  
    // ...  
}
```



C++17 Features - Code Simplify

Init-If

```
if (auto val = GetValue(); condition(val))  
{  
    // ...  
}
```

Zum Beispiel:

```
if (const auto it = myString.find("Hello");  
    it != std::string::npos)  
{  
    std::cout << it << " Hello\n";  
}
```



C++17 Features - Code Simplify

Init-If

```
if (auto val = GetValue(); condition(val))  
{  
    // ...  
}
```

Zum Beispiel:

```
if (const auto it = myString.find("Hello");  
    it != std::string::npos)  
{  
    std::cout << it << " Hello\n";  
}
```

- Scope verkleinern.



C++17 Features - Code Simplify

Structured Bindings [Live]

```
std::unordered_map<int, std::string> map;

if (auto [it, bSuccess] = map.emplace(3, "Banane"); bSuccess)
{
    std::cout << it->second; // dereferezieren
}
// `it` and `bSuccess` sind hier destruktiert.
```



C++17 Features - Code Simplify

Structured Bindings [Live]

```
std::unordered_map<int, std::string> map;

if (auto [it, bSuccess] = map.emplace(3, "Banane"); bSuccess)
{
    std::cout << it->second; // dereferezieren
}
// `it` and `bSuccess` sind hier destruktiert.
```



C++17 Features - Code Simplify

Structured Bindings [Live]

```
std::unordered_map<int, std::string> map;

if (auto [it, bSuccess] = map.emplace(3, "Banane"); bSuccess)
{
    std::cout << it->second; // dereferezieren
}
// `it` and `bSuccess` sind hier destruktiert.
```



C++17 Features - Code Simplify

Structured Bindings [Live]

```
std::unordered_map<int, std::string> map;

if (auto [it, bSuccess] = map.emplace(3, "Banane"); bSuccess)
{
    std::cout << it->second; // dereferezieren
}
// `it` and `bSuccess` sind hier destruktiert.
```



C++17 Features - Code Simplify

Structured Bindings [Live]

```
std::unordered_map<int, std::string> map;

if (auto [it, bSuccess] = map.emplace(3, "Banane"); bSuccess)
{
    std::cout << it->second; // dereferezieren
}
// `it` and `bSuccess` sind hier destruktiert.
```



C++17 Features - Code Simplify

Verschachteln von Namespace

```
namespace A::B::C {  
    // ...  
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



C++17 Features - Code Simplify

if constexpr

Compile-Time Switches:

```
template<typename ElementRef>
void compute(ElementRef ref)
{
    if constexpr(std::is_same_v<ElementRef, CVertexRef>) {
        // kompiliere das, was speziell ist für CVertexRef
    }
    else if constexpr(std::is_same_v<ElementRef, TriangleRef>) {
        // kompiliere das, was speziell ist für TriangleRef
    }
    else {
        static_assert(false, "Not-implemented!");
    }
}
```



Teil 2





Teil 2





C++ → 8a ?





Es geht anscheinend noch schwieriger: MUMPS

```
%DTC ; SF/XAK - DATE/TIME OPERATIONS ;1/16/92 11:36 AM
;; 19.0;VA FileMan;; Jul 14, 1992
D   I 'X1!'X2 S X="" Q
S X=X1 D H S X1=%H,X=X2,X2=%Y+1 D H S X=X1-%H,%Y=%Y+1&X2
K %H,X1,X2 Q
;
C   S X=X1 Q:'X D H S %H=%H+X2 D YMD S:$P(X1,".",2)
S S %=%#60/100+(%#3600\60)/100+(%\3600)/100 Q
;
H   I X<1410000 S %H=0,%Y=-1 Q
S %Y=$E(X,1,3),%M=$E(X,4,5),%D=$E(X,6,7)
S %T=$E(X_0,9,10)*60+$E(X_"000",11,12)*60+$E(X_"00000",13,14)
TOH S %H=%M>2&'(%Y#4)+$P("^31^59^90^120^151^181^212^243", "^", %M)
```



Motivation

Injektion der Werte eines Tuples

```
std::tuple<int, double, ... , std::string> t(1, 4.0, ... , "kiwi");
```

in eine Funktion f(...)

```
f(1, 4.0, ... , "kiwi");
```



Motivation

Injektion der Werte eines Tuples

```
std::tuple<int, double, ... , std::string> t(1, 4.0, ... , "kiwi");
```

in eine Funktion f(...)

```
f(1, 4.0, ... , "kiwi");
```

Rezept



Motivation

Injektion der Werte eines Tuples

```
std::tuple<int, double, ... , std::string> t(1, 4.0, ... , "kiwi");
```

in eine Funktion `f(...)`

```
f(1, 4.0, ... , "kiwi");
```

Rezept

- 500gr *Variadische Templates*,



Motivation

Injektion der Werte eines Tuples

```
std::tuple<int, double, ... , std::string> t(1, 4.0, ... , "kiwi");
```

in eine Funktion `f(...)`

```
f(1, 4.0, ... , "kiwi");
```

Rezept

- 500gr *Variadische Templates*,
- 1 EL *Perfect-Forwarding*,



Motivation

Injektion der Werte eines Tuples

```
std::tuple<int, double, ... , std::string> t(1, 4.0, ... , "kiwi");
```

in eine Funktion `f(...)`

```
f(1, 4.0, ... , "kiwi");
```

Rezept

- 500gr *Variadische Templates*,
- 1 EL *Perfect-Forwarding*,
- 2 Stück *Template Lambdas*,



Motivation

Injektion der Werte eines Tuples

```
std::tuple<int, double, ... , std::string> t(1, 4.0, ... , "kiwi");
```

in eine Funktion `f(...)`

```
f(1, 4.0, ... , "kiwi");
```

Rezept

- 500gr *Variadische Templates*,
- 1 EL *Perfect-Forwarding*,
- 2 Stück *Template Lambdas*,
- 1 x *Traits*



Motivation

Injektion der Werte eines Tuples

```
std::tuple<int, double, ... , std::string> t(1, 4.0, ... , "kiwi");
```

in eine Funktion `f(...)`

```
f(1, 4.0, ... , "kiwi");
```

Rezept

- 500gr *Variadische Templates*,
- 1 EL *Perfect-Forwarding*,
- 2 Stück *Template Lambdas*,
- 1 x *Traits*

Rühren und Kneten → Resultat 14-Zeilen am Schluss.



Perfect-Forwarding

Was kann man mit **forwarding**-Referenzen `T&&` [Template-Parameter T] erreichen:

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1, 2, 3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```



Perfect-Forwarding

Was kann man mit **forwarding**-Referenzen `T&&` [Template-Parameter T] erreichen:

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1, 2, 3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` wird kopiert ☺



Perfect-Forwarding

Was kann man mit **forwarding**-Referenzen `T&&` [Template-Parameter T] erreichen:

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1, 2, 3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` wird kopiert ☺
- `Triangles{{1,2,3}}` wird auch kopiert obwohl **rvalue** und könnte gemoved werden. 😞



Perfect-Forwarding

Was kann man mit **forwarding**-Referenzen `T&&` [Template-Parameter T] erreichen:

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1, 2, 3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` wird kopiert ☺
- `Triangles{{1,2,3}}` wird auch kopiert obwohl **rvalue** und könnte gemoved werden. 😞
- Lets fix it ...



Perfect-Forwarding

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Triangulation::Triangulation(const Vertices& v, Triangles&& t)
    : m_vertices(v), m_triangles(std::move(t))
{ /* code duplikation + std::move( ... ) */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```



Perfect-Forwarding

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }
```

```
Triangulation::Triangulation(const Vertices& v, Triangles&& t)
    : m_vertices(v), m_triangles(std::move(t))
{ /* code duplikation + std::move( ... ) */ }
```

```
Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```



Perfect-Forwarding

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Triangulation::Triangulation(const Vertices& v, Triangles&& t)
    : m_vertices(v), m_triangles(std::move(t))
{ /* code duplikation + std::move( ... ) */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` wird kopiert ☺



Perfect-Forwarding

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Triangulation::Triangulation(const Vertices& v, Triangles&& t)
    : m_vertices(v), m_triangles(std::move(t))
{ /* code duplikation + std::move( ... ) */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
    Triangles{{1,2,3}})
```

- `vertices` wird kopiert ☺
- `Triangles{{1,2,3}}` wird gemoved. ☺



Perfect-Forwarding

```
Triangulation::Triangulation(const Vertices& v, const Triangles& t)
    : m_vertices(v), m_triangles(t)
{ /* initialisiere Zeugs ... */ }

Triangulation::Triangulation(const Vertices& v, Triangles&& t)
    : m_vertices(v), m_triangles(std::move(t))
{ /* code duplikation + std::move( ... ) */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` wird kopiert 🤦
- `Triangles{{1,2,3}}` wird gemoved. 🤦
- Code Duplikation 😵 → Kombinationshölle 😵



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }
```

```
Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue**



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- **vertices** ist **lvalue** → Compiler deduziert **V := Vertices&**



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue** → Compiler deduziert `V := Vertices&`
→ `std::forward` retourniert `Vertices& (lvalue)`



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue** → Compiler deduziert `V := Vertices&`
→ `std::forward` retourniert `Vertices& (lvalue)` → Kopie. ☺



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue** → Compiler deduziert `V := Vertices&`
→ `std::forward` retourniert `Vertices& (lvalue)` → Kopie. ⚡
- `Triangles{{1,2,3}}` ist **rvalue**



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue** → Compiler deduziert `V := Vertices&`
→ `std::forward` retourniert `Vertices& (lvalue)` → Kopie. ☺
- `Triangles{{1,2,3}}` ist **rvalue** → Compiler deduziert `T := Triangles`



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue** → Compiler deduziert `V := Vertices&`
→ `std::forward` retourniert `Vertices&` (**lvalue**) → Kopie. ☺
- `Triangles{{1,2,3}}` ist **rvalue** → Compiler deduziert `T := Triangles`
→ `std::forward` retourniert `Triangles&&` (**xvalue**)



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }

Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue** → Compiler deduziert `V := Vertices&`
→ `std::forward` retourniert `Vertices&` (**lvalue**) → Kopie. ☺
- `Triangles{{1,2,3}}` ist **rvalue** → Compiler deduziert `T := Triangles`
→ `std::forward` retourniert `Triangles&&` (**xvalue**) → Move. ☺



Perfect-Forwarding mit T&&

forwarding-Referenzen verwenden welche die **Value-Kategorie** erhalten:

```
template<typename V, typename T>
Triangulation::Triangulation(V&& v, T&& t)
    : m_vertices(std::forward<V>(v))
    , m_triangles(std::forward<T>(t))
{ /* initialisiere Zeugs ... */ }
```

```
Vertices vertices = {1,2,3};
Triangulation tri(vertices,
                  Triangles{{1,2,3}})
```

- `vertices` ist **lvalue** → Compiler deduziert `V := Vertices&`
→ `std::forward` retourniert `Vertices& (lvalue)` → Kopie. ☺
- `Triangles{{1,2,3}}` ist **rvalue** → Compiler deduziert `T := Triangles`
→ `std::forward` retourniert `Triangles&& (xvalue)` → Move. ☺
- Keine Code-Duplikation. ☺

Template Refresher



Template Refresher



- Klassen/Funktionen

```
template<typename Iterable>
void findZero(const Iterable& it){ ... }
```

Template Refresher



- Klassen/Funktionen

```
template<typename Iterable>
void findZero(const Iterable& it){ ... }
```

- Lambdas

```
[ ](auto v){ std::cout << v; } // auto ⇔ typename T
```



Template Refresher

- Klassen/Funktionen

```
template<typename Iterable>
void findZero(const Iterable& it){ ... }
```

- Lambdas

```
[ ](auto v){ std::cout << v; } // auto ⇔ typename T
```

- **Typedefinitionen:** Alias-Templates:

```
template<typename T>
using Map = std::map<int, T*>; // → Map<float> a;
```

Template Refresher



- Klassen/Funktionen

```
template<typename Iterable>
void findZero(const Iterable& it){ ... }
```

- Lambdas

```
[](auto v){ std::cout << v; } // auto ⇔ typename T
```

- **Typedefinitionen:** Alias-Templates:

```
template<typename T>
using Map = std::map<int, T*>; // → Map<float> a;
```

Don't: Old-School über `typedef` und `struct` → unleserlich:

```
template<typename T>
struct Trait { typedef std::map<int, T*> type; };
Trait<float>::type; // std::map<int, float*>
```



Nomenklatur



Nomenklatur

- Template Parameter

```
//                                     vvvv----- Template Parameter
template<typename Type>
void print(Type v);
```



Nomenklatur

- Template Parameter

```
//           vvvv----- Template Parameter
template<typename Type>
void print(Type v);
```

- Template Argument

```
//     vvv----- Template Argument
print<int>(3);
```



Nomenklatur

- Template Parameter

```
//           vvvv----- Template Parameter
template<typename Type>
void print(Type v);
```

- Template Argument

```
//     vvv----- Template Argument
print<int>(3);
```

- `std :: vector<int>` ist ein **Typ**.



Nomenklatur

- Template Parameter

```
//           vvvv----- Template Parameter
template<typename Type>
void print(Type v);
```

- Template Argument

```
//     vvv----- Template Argument
print<int>(3);
```

- `std :: vector<int>` ist ein **Typ**.
- `std :: vector` ist kein Typ sondern ein **Template** mit 2 Template-Parameter.
Im Beispiel: Type kann **nie** `std :: vector` sein!



Nomenklatur

- Template Parameter

```
//           vvvv----- Template Parameter
template<typename Type>
void print(Type v);
```

- Template Argument

```
//     vvv----- Template Argument
print<int>(3);
```

- `std::vector<int>` ist ein **Typ**.
- `std::vector` ist kein Typ sondern ein **Template** mit 2 Template-Parameter.
Im Beispiel: Type kann **nie** `std::vector` sein!
- **Traits** sind Klassentemplates mit *intern* definierten Typen: [`type_traits`]

```
template<typename T> struct std::add_pointer{ using type = T*; };
```



Static Asserts

```
#include <type_traits>
static_assert(std::is_same<DataType, CVertexRef>::value, "Wups!");
static_assert(std::is_same<DataType, CVertexRef>{}, "Wups!");
```



Static Asserts

```
#include <type_traits>
static_assert(std::is_same<DataType, CVertexRef>::value, "Wups!");
static_assert(std::is_same<DataType, CVertexRef>{}, "Wups!");
```



Static Asserts

```
#include <type_traits>
static_assert(std::is_same<DataType, CVertexRef>::value, "Wups!");
static_assert(std::is_same<DataType, CVertexRef>{}, "Wups!");
```



Static Asserts

```
#include <type_traits>
static_assert(std::is_same<DataType, CVertexRef>::value, "Wups!");
static_assert(std::is_same<DataType, CVertexRef>{}, "Wups!");
```

- C++14 : Variable Template

```
static_assert(std::is_same_v<DataType, CVertexRef>, "Wups!");
```

```
template<typename T, typename U>
constexpr bool is_same_v = std::is_same<T, U>::value;
```



Static Asserts

```
#include <type_traits>
static_assert(std::is_same<DataType, CVertexRef>::value, "Wups!");
static_assert(std::is_same<DataType, CVertexRef>{}, "Wups!");
```

- C++14 : Variable Template

```
static_assert(std::is_same_v<DataType, CVertexRef>, "Wups!");
```

```
template<typename T, typename U>
constexpr bool is_same_v = is_same<T, U>::value;
```

Template Parameter Arten



Template Parameter Arten



- Type Template-Parameter

```
template<typename T> struct A{}; // `class T` → das selbe (dont!)
```



Template Parameter Arten

- Type Template-Parameter

```
template<typename T> struct A{}; // `class T` → das selbe (dont!)
```

- Non-Type Template-Parameter

```
template<std::size_t N> struct A{};  
template<MyEnum EMode> struct B{};  
template<auto N> struct C{}; // C++20
```



Template Parameter Arten

- Type Template-Parameter

```
template<typename T> struct A{}; // `class T` → das selbe (dont!)
```

- Non-Type Template-Parameter

```
template<std::size_t N> struct A{};  
template<MyEnum EMode> struct B{};  
template<auto N> struct C{}; // C++20
```



Template Parameter Arten

- Type Template-Parameter

```
template<typename T> struct A{}; // `class T` → das selbe (dont!)
```

- Non-Type Template-Parameter

```
template<std::size_t N> struct A{};  
template<MyEnum EMode> struct B{};  
template<auto N> struct C{}; // C++20
```



Template Parameter Arten

- Type Template-Parameter

```
template<typename T> struct A{}; // `class T` → das selbe (dont!)
```

- Non-Type Template-Parameter

```
template<std::size_t N> struct A{};  
template<MyEnum EMode> struct B{};  
template<auto N> struct C{}; // C++20
```



Template Parameter Arten

- Type Template-Parameter

```
template<typename T> struct A{}; // `class T` → das selbe (dont!)
```

- Non-Type Template-Parameter

```
template<std::size_t N> struct A{};  
template<MyEnum EMode> struct B{};  
template<auto N> struct C{}; // C++20
```

Don't. Template Argument-Matching schwierig bis unmöglich!

Wappe alles in Typen, z.B.:

```
std::integral_constant<int, 3> mit typename Number
```



Template Parameter Arten

- Type Template-Parameter

```
template<typename T> struct A{}; // `class T` → das selbe (dont!)
```

- Non-Type Template-Parameter

```
template<std::size_t N> struct A{};  
template<MyEnum EMode> struct B{};  
template<auto N> struct C{}; // C++20
```

Don't. Template Argument-Matching schwierig bis unmöglich!

Wappe alles in Typen, z.B.:

```
std::integral_constant<int, 3> mit typename Number
```

- Template-Template-Parameter

```
template<template<typename> class T> struct A{};
```

Don't. Das will man immer vermeiden! Es gibt bessere Konzepte (siehe Alias/Callables [meta]).



Variadische Parameter [Live]

```
template<typename ... Types>
class Converter{ // ^^^^^----- Parameter-Pack
public:
    using Tuple = std::tuple<Types ... >;
    // ^^^----- Pack-Expansion

private:
    Tuple m_tuple;
    std::array<int, sizeof...(Types)> m_count;
    // ^^^^^^^^^----- Anzahl Parameter
};

Converter<int, float, double> c; // 1. `Tuple` und `m_count`?
```



Variadische Parameter [Live]

```
template<typename ... Types>
class Converter{ // ^^^^^----- Parameter-Pack
public:
    using Tuple = std::tuple<Types ... >;
    // ^^^----- Pack-Expansion

private:
    Tuple m_tuple;
    std::array<int, sizeof...(Types)> m_count;
    // ^^^^^^^^^----- Anzahl Parameter
};

Converter<int, float, double> c; // 1. `Tuple` und `m_count`?
```



Variadische Parameter [Live]

```
template<typename ... Types>
class Converter{ // ^^^^^----- Parameter-Pack
public:
    using Tuple = std::tuple<Types ... >;
    // ^^^----- Pack-Expansion

private:
    Tuple m_tuple;
    std::array<int, sizeof...(Types)> m_count;
    // ^^^^^^^^^----- Anzahl Parameter
};

Converter<int, float, double> c; // 1. `Tuple` und `m_count`?
```



Variadische Parameter [Live]

```
template<typename ... Types>
class Converter{ // ^^^^^----- Parameter-Pack
public:
    using Tuple = std::tuple<Types ... >; // ^^^----- Pack-Expansion

private:
    Tuple m_tuple;
    std::array<int, sizeof...(Types)> m_count; // ^^^^^^^^^----- Anzahl Parameter
};

Converter<int, float, double> c; // 1. `Tuple` und `m_count`?
```



Variadische Parameter [Live]

```
template<typename ... Types>
class Converter{ // ^^^^^----- Parameter-Pack
public:
    using Tuple = std::tuple<Types ... >;
    // ^^^----- Pack-Expansion

private:
    Tuple m_tuple;
    std::array<int, sizeof...(Types)> m_count;
    // ^^^^^^^^^----- Anzahl Parameter
};

Converter<int, float, double> c; // 1. `Tuple` und `m_count`?
```



Variadische Parameter [Live]

```
template<typename ... Types>
class Converter{//
public:
    using Tuple = std::tuple<Types ... >;
    //                                         ^^^----- Pack-Expansion

private:
    Tuple m_tuple;
    std::array<int, sizeof...(Types)> m_count;
    //                                         ^^^^^^----- Anzahl Parameter
};

Converter<int, float, double> c; // 1. `Tuple` und `m_count`?
```

- `Tuple := std::tuple<int, float, double>`



Variadische Parameter [Live]

```
template<typename ... Types>
class Converter{//
public:
    using Tuple = std::tuple<Types ... >;
    //

private:
    Tuple m_tuple;
    std::array<int, sizeof...(Types)> m_count;
    //
};

Converter<int, float, double> c; // 1. `Tuple` und `m_count`?
```

- `Tuple := std::tuple<int, float, double>`
- `m_count := std::array<int, 3>`



Variadische Parameter [Live]

Meta-Programming: Rechnen mit Typen zu Kompilierzeit:

```
using List      = meta::list<double, float, int>;
using ListNew = meta::transform<List,
                                meta::quote<std::add_pointer_t>>;
ListNew::DJBobo;
```

```
error: 'DJBobo' is not a member of
'ListNew' {aka 'meta::list<double*, float*, int*>'}
```



Reference-Collapsing Regeln

```
foo<int&>();  
  
template<typename T>  
void foo()  
{  
    const T& temp = 3; // 'T' ist 'int&'  
                      // 'temp' ist 'const (int&) &' → ☺ ☺  
}
```



Reference-Collapsing Regeln

```
foo<int&>();  
  
template<typename T>  
void foo()  
{  
    const T& temp = 3; // 'T' ist 'int&'  
                      // 'temp' ist 'const (int&) &' → ☺ ☺  
}
```



Reference-Collapsing Regeln

```
foo<int&>();  
  
template<typename T>  
void foo()  
{  
    const T& temp = 3; // 'T' ist 'int&'  
                      // 'temp' ist 'const (int&) &' → ⓘ ⓘ  
}
```



Reference-Collapsing Regeln

```
foo<int&>();  
  
template<typename T>  
void foo()  
{  
    const T& temp = 3; // 'T' ist 'int&'  
                      // 'temp' ist 'const (int&) &' → ⓘ ⓘ  
}
```

Deshalb:

- $(T\&)$ $\&$ → kollabiert zu $T\&$
- $(T\&)$ $\&\&$ → kollabiert zu $T\&$
- $(T\&\&)$ $\&$ → kollabiert zu $T\&$
- $(T\&\&)$ $\&\&$ → kollabiert zu $T\&\&$



Reference-Collapsing Regeln

```
foo<int&>();  
  
template<typename T>  
void foo()  
{  
    const T& temp = 3; // 'T' ist 'int&'  
                      // 'temp' ist 'const (int&) &' → ☺ ☺  
}
```

Deshalb:

- $(T\&)$ & → kollabiert zu $T\&$
- $(T\&)$ && → kollabiert zu $T\&$
- $(T\&\&)$ & → kollabiert zu $T\&$
- $(T\&\&)$ && → kollabiert zu $T\&\&$

Eselstrücke: Einfache Referenz & gewinnt immer.



Quiz

```
template<typename T>
void foo(T& v);

int main()
{
    int&& a = 3;
    foo<int&&>(a); // 1. Typ von `v`.
}
```



Quiz

```
template<typename T>
void foo(T& v);

int main()
{
    int&& a = 3;
    foo<int&&>(a); // 1. Typ von `v`.
}
```



Quiz

```
template<typename T>
void foo(T& v);

int main()
{
    int&& a = 3;
    foo<int&&>(a); // 1. Typ von `v`.
}
```

1. $T\& := (\text{int}\&\&) \quad \& := \text{int}\&$



Template Argument Deduction

[temp.deduct.call]

```
Banane c;  
add(c);  
// ^----- Argument Typ → trafo → definiert Typ `A`  
  
template<typename T>  
void add(const T& val);  
// ^^^^^^---- Parameter Typ → trafo → definiert Typ `P`
```



Template Argument Deduction

[temp.deduct.call]

```
Banane c;  
add(c);  
// ^----- Argument Typ → trafo → definiert Typ `A`  
  
template<typename T>  
void add(const T& val);  
// ^^^^^^----- Parameter Typ → trafo → definiert Typ `P`
```

Template Parameter **T** wird vom Compiler automatisch **deduziert**. Der Kompiler arbeitet mit 2 Typen **A** und **P**.



Template Argument Deduction

[temp.deduct.call]

```
Banane c;  
add(c);  
// ^----- Argument Typ → trafo → definiert Typ `A`  
  
template<typename T>  
void add(const T& val);  
// ^^^^^^----- Parameter Typ → trafo → definiert Typ `P`
```

Template Parameter **T** wird vom Compiler automatisch **deduziert**. Der Kompiler arbeitet mit 2 Typen **A** und **P**.

- **A** := Banane



Template Argument Deduction

[temp.deduct.call]

```
Banane c;  
add(c);  
// ^----- Argument Typ → trafo → definiert Typ `A`  
  
template<typename T>  
void add(const T& val);  
// ^^^^^^---- Parameter Typ → trafo → definiert Typ `P`
```

Template Parameter **T** wird vom Compiler automatisch **deduziert**. Der Kompiler arbeitet mit 2 Typen **A** und **P**.

- **A** := Banane
- **P** := **T**



Template Argument Deduction

[temp.deduct.call]

```
Banane c;  
add(c);  
// ^----- Argument Typ → trafo → definiert Typ `A`  
  
template<typename T>  
void add(const T& val);  
// ^^^^^^---- Parameter Typ → trafo → definiert Typ `P`
```

Template Parameter **T** wird vom Compiler automatisch **deduziert**. Der Kompiler arbeitet mit 2 Typen **A** und **P**.

- **A** := Banane
- **P** := **T**

Matche **A** mit **P**: → **Resultat:** **T** := Banane



Template Argument Deduction

[temp.deduct.call]

```
Banane c;  
add(c);  
// ^----- Argument Typ → trafo → definiert Typ `A`  
  
template<typename T>  
void add(const T& val);  
// ^^^^^^---- Parameter Typ → trafo → definiert Typ `P`
```

Template Parameter **T** wird vom Compiler automatisch **deduziert**. Der Kompiler arbeitet mit 2 Typen **A** und **P**.

- **A** := Banane
- **P** := **T**

Matche **A** mit **P**: → **Resultat:** **T** := Banane

Was sind die Regeln und wie wird gematched?



Template Argument Deduction

[temp.deduct.call]

Es gibt genau drei Unterscheidungs-Fälle:



Template Argument Deduction

[temp.deduct.call]

Es gibt genau drei Unterscheidungs-Fälle:

- Deklaration T :

```
template<typename T> void add(T val);           // T ⇔ auto
```



Template Argument Deduction

[temp.deduct.call]

Es gibt genau drei Unterscheidungs-Fälle:

- Deklaration `T`:

```
template<typename T> void add(T val);           // T ⇔ auto
```

- Deklaration `T&` (**lvalue**-Referenz):

```
template<typename T> void add(      T& val); // T& ⇔ auto&
template<typename T> void add(const T& val); // T& ⇔ const auto&
```



Template Argument Deduction

[temp.deduct.call]

Es gibt genau drei Unterscheidungs-Fälle:

- Deklaration `T`:

```
template<typename T> void add(T val);           // T ⇔ auto
```

- Deklaration `T&` (**lvalue**-Referenz):

```
template<typename T> void add(      T& val); // T& ⇔ auto&
template<typename T> void add(const T& val); // T& ⇔ const auto&
```

- Deklaration `T&&` (**forwarding**-Referenz):

```
template<typename T> void add(T&& val);     // T&& ⇔ auto&&
```

Deduktion bei T oder auto [temp.deduct.call]



```
const Banane& c = ... ;  
add(c);  
// ^----- [entferne const, etc ... ] → A := Banane
```

```
template<typename T>  
void add(T val);  
// ^----- → P := T
```

Wichtig: T wird nie automatisch zu einer Referenz. add<int&>(3) ist keine automatische Deduktion!

Deduktion bei T oder auto [temp.deduct.call]



```
const Banane& c = ... ;  
add(c);  
// ^----- [entferne const, etc ... ] → A := Banane
```

```
template<typename T>  
void add(T val);  
// ^----- → P := T
```

- **Argument-Typ A:**

Expr. `c`: *const Banane* [7.2.2#1]

Trafos:

- Entferne `const` von `A`. [13.9.2.1#2.3]
- und noch andere unwichtige Traforegeln auf `A`.

Wichtig: `T` wird nie automatisch zu einer Referenz. `add<int&>(3)` ist keine **automatische** Deduktion!

Deduktion bei T oder auto [temp.deduct.call]



```
const Banane& c = ... ;  
add(c);  
// ^----- [entferne const, etc ... ] → A := Banane
```

```
template<typename T>  
void add(T val);  
// ^----- → P := T
```

- **Argument-Typ A:**

Expr. `c`: *const Banane* [7.2.2#1]

Trafos:

- Entferne `const` von `A`. [13.9.2.1#2.3]
- und noch andere unwichtige Traforegeln auf `A`.

- **Parameter-Typ P:**

Trafos: keine. `P` wird zu `T`. [13.9.2.1#2 impli.]

Wichtig: `T` wird nie automatisch zu einer Referenz. `add<int&>(3)` ist keine **automatische** Deduktion!

Deduktion bei T oder auto [temp.deduct.call]



```
const Banane& c = ... ;
add(c);
// ^----- [entferne const, etc ... ] → A := Banane
```

```
template<typename T>
void add(T val);
// ^----- → P := T
```

- **Argument-Typ A:**

Expr. `c`: *const Banane* [7.2.2#1]

Trafos:

- Entferne `const` von `A`. [13.9.2.1#2.3]
- und noch andere unwichtige Traforegeln auf `A`.

- **Parameter-Typ P:**

Trafos: keine. `P` wird zu `T`. [13.9.2.1#2 impli.]

- **Pattern-Match:** `P` mit `A` ergibt `T := Banane`

Wichtig: `T` wird nie automatisch zu einer Referenz. `add<int&>(3)` ist keine **automatische** Deduktion!



Deduktion bei **T&** oder **auto&** [temp.deduct.call]

```
const Banane& c = ... ;  
add(c);  
//   ^----- → A := const Banane
```

```
template<typename T>  
void add(T& val);           // (oder const T&)  
//   ^----- [entferne const und &] → P := T
```



Deduktion bei `T&` oder `auto&` [temp.deduct.call]

```
const Banane& c = ... ;  
add(c);  
//   ^----- → A := const Banane
```

```
template<typename T>  
void add(T& val);           // (oder const T&)  
//       ^^----- [entferne const und &] → P := T
```

- Argument-Typ A:

Expr. `c : const Banane` [7.2.2#1]

Trafos: keine. (`const` darf z.B. nicht entfernt werden!)



Deduktion bei `T&` oder `auto&` [temp.deduct.call]

```
const Banane& c = ... ;  
add(c);  
// ^----- → A := const Banane
```

```
template<typename T>  
void add(T& val);           // (oder const T&)  
// ^----- [entferne const und &] → P := T
```

- **Argument-Typ A:**

Expr. `c : const Banane` [7.2.2#1]

Trafos: keine. (`const` darf z.B. nicht entfernt werden!)

- **Parameter-Typ P:**

Trafos: Entferne `const` und Referenz `&` [13.9.2.1#3]



Deduktion bei `T&` oder `auto&` [temp.deduct.call]

```
const Banane& c = ... ;
add(c);
// ^----- → A := const Banane
```

```
template<typename T>
void add(T& val);           // (oder const T&)
//      ^^----- [entferne const und &] → P := T
```

- **Argument-Typ A:**

Expr. `c : const Banane` [7.2.2#1]

Trafos: keine. (`const` darf z.B. nicht entfernt werden!)

- **Parameter-Typ P:**

Trafos: Entferne `const` und Referenz `&` [13.9.2.1#3]

Pattern-Match: `P mit A.` (ergibt `T := const Banane`)



Deduktion bei `T&&` oder `auto&&` [temp.deduct.call]

- `T&&` [Template-Parameter T]
- `auto&&`

sind **forwarding**-Referenzen.

Bei der automatischen Deduktion können diese zu einer **lvalue**-Referenz (&) oder einer **rvalue**-Referenz (&&) werden!



Deduktion bei T&& oder auto&& [temp.deduct.call]

```
const Banane& c;  
add(c);  
// ^----- [lvalue → füge & hinzu] → A := const Banane&  
add(4);  
// ^----- [rvalue → nichts] → A := int  
  
template<typename T>  
void add(T&& val);  
//      ^^^----- [entferne const und &] → P := T
```



Deduktion bei T&& oder auto&& [temp.deduct.call]

```
const Banane& c;
add(c);
// ^----- [lvalue → füge & hinzu] → A := const Banane&
add(4);
// ^----- [rvalue → nichts]      → A := int

template<typename T>
void add(T&& val);
//     ^^^----- [entferne const und &] → P := T
```

- Argument-Typ A:

Trafos: [13.9.2.1#3]



Deduktion bei T&& oder auto&& [temp.deduct.call]

```
const Banane& c;
add(c);
// ^----- [lvalue → füge & hinzu] → A := const Banane&
add(4);
// ^----- [rvalue → nichts] → A := int

template<typename T>
void add(T&& val);
//     ^^^----- [entferne const und &] → P := T
```

- Argument-Typ A:

Trafos: [13.9.2.1#3]

- Expr. c ist [const Banane, lvalue] → Add & → A := const Banane&



Deduktion bei T&& oder auto&& [temp.deduct.call]

```
const Banane& c;
add(c);
// ^----- [lvalue → füge & hinzu] → A := const Banane&
add(4);
// ^----- [rvalue → nichts]      → A := int

template<typename T>
void add(T&& val);
//     ^^^----- [entferne const und &] → P := T
```

- **Argument-Typ A:**

Trafos: [13.9.2.1#3]

- Expr. `c` ist [const Banane, lvalue] → Add & → A := const Banane&
- Expr. `4` ist [int, prvalue] → A := int



Deduktion bei `T&&` oder `auto&&` [temp.deduct.call]

```
const Banane& c;
add(c);
// ^----- [lvalue → füge & hinzu] → A := const Banane&
add(4);
// ^----- [rvalue → nichts]      → A := int

template<typename T>
void add(T&& val);
//     ^^^----- [entferne const und &] → P := T
```

- **Argument-Typ A:**

Trafos: [13.9.2.1#3]

- Expr. `c` ist [const Banane, lvalue] → Add & → `A := const Banane&`
- Expr. `4` ist [int, prvalue] → `A := int`

- **Parameter-Typ P:**

Trafos: Siehe Deklaration `T&`. `P := T.`



Deduktion bei `T&&` oder `auto&&` [temp.deduct.call]

```
const Banane& c;
add(c);
// ^----- [lvalue → füge & hinzu] → A := const Banane&
add(4);
// ^----- [rvalue → nichts] → A := int

template<typename T>
void add(T&& val);
//     ^^^----- [entferne const und &] → P := T
```

- **Argument-Typ A:**

Trafos: [13.9.2.1#3]

- Expr. `c` ist [const Banane, lvalue] → Add & → `A := const Banane&`
- Expr. `4` ist [int, prvalue] → `A := int`

- **Parameter-Typ P:**

Trafos: Siehe Deklaration `T&`. `P := T.`

- **Pattern-Match:** `T := const Banane&, T := int` → Ref. Collapse!



Deklaration T&& [temp.deduct.call]

```
const Banane& a;  
add(a);  
// ^----- 1. → val := (const Banane&) && := const Banane&  
add(4);  
// ^----- 2. → val := (int) && := int&&  
  
template<typename T>  
void add(T&& val);
```



Deklaration T&& [temp.deduct.call]

```
const Banane& a;
add(a);
// ^----- 1. → val := (const Banane&) && := const Banane&
add(4);
// ^----- 2. → val := (int) && := int&&

template<typename T>
void add(T&& val);
```

1. **lvalue** wird weitergegeben als **lvalue**-Reference → ⚡



Deklaration T&& [temp.deduct.call]

```
const Banane& a;  
add(a);  
// ^----- 1. → val := (const Banane&) && := const Banane&  
add(4);  
// ^----- 2. → val := (int) && := int&&  
  
template<typename T>  
void add(T&& val);
```

1. **lvalue** wird weitergegeben als **lvalue**-Reference → ↗
2. **rvalue** wird weitergegeben als **rvalue**-Reference → ↗



Deklaration T&& [temp.deduct.call]

```
template<typename T>
void add(T&& val)
{
    add(val); // Ungut: `val` wird immer als lvalue weitergegeben
}
```

- `val` ist **lvalue** und zweideutiger Typ: **rvalue/lvalue**-Reference.



Deklaration T&& [temp.deduct.call]

```
template<typename T>
void add(T&& val)
{
    add(val); // Ungut: `val` wird immer als lvalue weitergegeben
}
```

- `val` ist **lvalue** und zweideutiger Typ: **rvalue/lvalue**-Reference.

Merke: Um die Zweideutigkeit von `T&&` zu erhalten, braucht's immer `std::forward<T>`:

```
template<typename T>
void add(T&& val)
{
    add(std::forward<T>(val)); // Richtig!
}
```



Quiz [Live]

Was ist der Typ von t ?

```
Banane&& a = 4;  
foo(a);  
//   ^----- A := ???
```

```
template<typename T>  
void foo(T&& t);  
//      ??? ----- P := ???
```



Quiz [Live]

Was ist der Typ von `t` ?

```
Banane&& a = 4;  
foo(a);  
//   ^----- A := ???
```

```
template<typename T>  
void foo(T&& t);  
//      ??? ----- P := ???
```

- Für `A` : Expression `a` ist `[Banane, lvalue]` → add `&` → `A := Banane&`



Quiz [Live]

Was ist der Typ von t ?

```
Banane&& a = 4;  
foo(a);  
// ^----- A := ???
```

```
template<typename T>  
void foo(T&& t);  
// ??? ----- P := ???
```

- Für A : Expression a ist [Banane, lvalue] → add & → A := Banane&
- Für P : → entferne const/& → P := T.



Quiz [Live]

Was ist der Typ von `t` ?

```
Banane&& a = 4;  
foo(a);  
//   ^----- A := ???
```

```
template<typename T>  
void foo(T&& t);  
//      ??? ----- P := ???
```

- Für `A` : Expression `a` ist `[Banane, lvalue]` → add `&` → `A := Banane&`
- Für `P` : → entferne `const/&` → `P := T`.
- Pattern-Match: → `T := Banane&` → Ref. Coll. → `T&& := Banane&`



Wie funktioniert std :: forward<T>

Zwei Overloads:

1. Um **lvalues** als **lvalues/rvalues** zu forwarden (abh. von T)

```
template<typename T>
T&& forward(lvalue-Reference v){
    return static_cast<T&&>(v);
};
```

2. Um **rvalues** nur als **rvalues** zu forwarden (Esoterische Cases)

```
template<typename T>
T&& forward(rvalue-Reference v){
    // static-assert: T keine lvalue-Reference
    return static_cast<T&&>(v);
};
```

[[Live](#)] Good Read: [N2951](#)



Benutzen von `std :: forward`

Merke: `std :: forward` geht immer zusammen mit einer forwarding-Referenz `T&&` !

```
template<typename T>;
struct Shake {
    void doIt(T&& fruit);
    //           ^^^ ----- keine forwarding-Referenz,
    //                               da `T` hier nicht mehr deduziert wird!
}
```



Benutzen von `std::forward`

Merke: `std::forward` geht immer zusammen mit einer forwarding-Referenz `T&&`!

```
template<typename T>
struct Shake {
    void doIt(T&& fruit);
    //           ^^^ ----- keine forwarding-Referenz,
    //                   da `T` hier nicht mehr deduziert wird!
}
```

```
template<typename T>
struct Shake {
    template<typename F>
    void doIt(F&& fruit);
    //           ^^^ ----- forwarding-Referenz, da deduziert!
    //                   ('const F&&' wäre keine)
}
```



Zusammenfassung std :: move

- `std :: move(expr)` wandelt die Expression `expr` zu einer **rvalue** um (retourniert ein *xvalue*). Man macht diesen Cast um anzugeben, dass man sich nicht länger um den Wert von `expr` kümmert **nach der Auswertung der Expression** in welchem dieser Wert gebraucht wurde. Zum Beispiel:

```
y = std::move(x);
// `y` hat den Wert von `x` und `x` interessiert uns nicht mehr
x = getNewValue(); // weil wir (optional) einen neuen Wert zuweisen.
```

Merke: Wann immer man eine **rvalue**-Referenz hat (e.g. Banane&& rB, d.h. es geht um temporäre Objekte), kommt `std :: move` zur Anwendung.



Zusammenfassung std :: forward

- `std :: forward<T>(expr)` is ähnlich zu `std :: move(expr)` und kann zu einem **rvalue** umwandeln. Es hat jedoch zwei Eingaben: `expr` und `T`. `T` wird benutzt um zu entscheiden ob ein **lvalue** oder ein **rvalue** retourniert wird. Wenn `T` eine **lvalue**-Referenz ist, dann wird eine **lvalue**-Referenz (`&`) auf `expr` retourniert ansonsten eine **rvalue**-Referenz (`&&`).

```
struct A{} x,y,z;  
y = std::forward<A&>(x); // `x` wird nach `y` kopiert  
z = std::forward<A>(x); // `x` wird nach `z` gemoved.
```

Merke: Wann immer man eine **forwarding**-Referenz hat:

```
template<typename T> foo(T&& rB)
```

kommt `std :: forward<T>(rB)` zur Anwendung.



Beispiel 6: std::make_unique

```
Kiwi k;  
auto spShake = std::make_unique<Shake>(Banane{"mushy"}, k);  
  
template<typename T, typename ... Args>  
auto make_unique(Args&& ... args)  
{  
    return std::unique_ptr<T>(new T(std::forward<Args>(args) ... ));  
}
```



Beispiel 6: std::make_unique

```
Kiwi k;
auto spShake = std::make_unique<Shake>(Banane{"mushy"}, k);

template<typename T, typename ... Args>
auto make_unique(Args&& ... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args) ...));
}
```



Beispiel 6: std::make_unique

```
Kiwi k;  
auto spShake = std::make_unique<Shake>(Banane{"mushy"}, k);  
  
template<typename T, typename ... Args>  
auto make_unique(Args&& ... args)  
{  
    return std::unique_ptr<T>(new T(std::forward<Args>(args) ... ));  
}
```

Alle variadischen Argumente `args` werden per **Perfect-Forwarding** an den Konstruktor von `T` übergeben:



Beispiel 6: std::make_unique

```
Kiwi k;  
auto spShake = std::make_unique<Shake>(Banane{"mushy"}, k);  
  
template<typename T, typename ... Args>  
auto make_unique(Args&& ... args)  
{  
    return std::unique_ptr<T>(new T(std::forward<Args>(args) ... ));  
}
```

Alle variadischen Argumente `args` werden per **Perfect-Forwarding** an den Konstruktor von `T` übergeben:

- `Banane{"mushy"}` wird gemoved.



Beispiel 6: std::make_unique

```
Kiwi k;
auto spShake = std::make_unique<Shake>(Banane{"mushy"}, k);

template<typename T, typename ... Args>
auto make_unique(Args&& ... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args) ...));
}
```

Alle variadischen Argumente `args` werden per **Perfect-Forwarding** an den Konstruktor von `T` übergeben:

- `Banane{"mushy"}` wird gemoved.
- `Kiwi k` wird kopiert.



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };  
Functor func;  
  
template<typename F>  
void apply(int a, F&& f)  
{  
    f(a, 10);  
}  
  
apply(5, func);                                // 1. `F&&` → `Functor&`  
apply(5, createFunctor());                      // 2. `F&&` → `Functor&&`  
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };  
Functor func;  
  
template<typename F>  
void apply(int a, F&& f)  
{  
    f(a, 10);  
}  
  
apply(5, func);                                // 1. `F&&` → `Functor&`  
apply(5, createFunctor());                      // 2. `F&&` → `Functor&&`  
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };
```

```
Functor func;
```

```
template<typename F>
void apply(int a, F&& f)
{
    f(a, 10);
}
```

```
apply(5, func); // 1. `F&&` → `Functor&`
```

```
apply(5, createFunctor()); // 2. `F&&` → `Functor&&`
```

```
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };  
Functor func;  
  
template<typename F>  
void apply(int a, F&& f)  
{  
    f(a, 10);  
}  
  
apply(5, func);                                // 1. `F&&` → `Functor&`  
apply(5, createFunctor());                      // 2. `F&&` → `Functor&&`  
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };  
Functor func;  
  
template<typename F>  
void apply(int a, F&& f)  
{  
    f(a, 10);  
}  
  
apply(5, func);                                // 1. `F&&` → `Functor&`  
apply(5, createFunctor());                      // 2. `F&&` → `Functor&&`  
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };  
Functor func;  
  
template<typename F>  
void apply(int a, F&& f)  
{  
    f(a, 10);  
}  
  
apply(5, func);                                // 1. `F&&` → `Functor&`  
apply(5, createFunctor());                      // 2. `F&&` → `Functor&&`  
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };  
Functor func;  
  
template<typename F>  
void apply(int a, F&& f)  
{  
    f(a, 10);  
}  
  
apply(5, func);                                // 1. `F&&` → `Functor&`  
apply(5, createFunctor());                      // 2. `F&&` → `Functor&&`  
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```

- **forwarding**-Reference: Vermeidet Codeduplikationen und erweitert die Anwendbarkeit!



Beispiel 7: Funktor applizieren

```
struct Functor{ void operator()(int a, int b){/* ... */} };  
Functor func;  
  
template<typename F>  
void apply(int a, F&& f)  
{  
    f(a, 10);  
}  
  
apply(5, func);                                // 1. `F&&` → `Functor&`  
apply(5, createFunctor());                      // 2. `F&&` → `Functor&&`  
apply(5, [](int& a, int b){ a += b; }); // 3. `F&&` → `XXXX&&`
```

- **forwarding**-Reference: Vermeidet Codeduplikationen und erweitert die Anwendbarkeit!
- Benutze `static_assert(...)` für gewisse Typen-Checks.



Fettnäpfchen bei T&&

```
struct Banane
{
    template<typename T>
    Banane(T&& value);

    Banane(const Banane& rBanane); // wird verdeckt!
};
```

Wird Probleme geben, da T&& so möglichst alles matched was man sich vorstellen kann:

- Der Copy-CTOR wird nie matchen können, weil T&& stärker!
- T&& irgendwie entfernen und Overloading verwenden oder falls alles nicht hilft
- SFINAE verwenden (später)



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
auto tuple = std::make_tuple(1, 2.0, "Banane");

invoke(tuple,
       [](int a, double b, const std::string& c)
{
    std::cout << a << ", " << b << ", " << c << std::endl;
});
```

Output:

```
1, 2.0, Banane
```



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
auto tuple = std::make_tuple(1, 2.0, "Banane");

invoke(tuple,
       [](int a, double b, const std::string& c)
{
    std::cout << a << ", " << b << ", " << c << std::endl;
});
```

Output:

```
1, 2.0, Banane
```



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
auto tuple = std::make_tuple(1, 2.0, "Banane");

invoke(tuple,
       [](int a, double b, const std::string& c)
       {
           std::cout << a << ", " << b << ", " << c << std::endl;
       });

```

Output:

```
1, 2.0, Banane
```



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
invoke(std::make_tuple(1, 2.0, "Banane"),
       [](int a, double b, const std::string& c){});
```

Wie erreichen wir das:



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
invoke(std::make_tuple(1, 2.0, "Banane"),
       [](int a, double b, const std::string& c){});
```

Wie erreichen wir das:

- Template Funktion `invoke`.



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
invoke(std::make_tuple(1, 2.0, "Banane"),
       [](int a, double b, const std::string& c){});
```

Wie erreichen wir das:

- Template Funktion `invoke`.
- **forwarding**-Referenzen brauchen, damit
`invoke(std::make_tuple(...))` möglich.



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
invoke(std::make_tuple(1, 2.0, "Banane"),
       [](int a, double b, const std::string& c){});
```

Wie erreichen wir das:

- Template Funktion `invoke`.
- **forwarding**-Referenzen brauchen, damit `invoke(std::make_tuple(...))` möglich.
- Meta-Programming möglichst **einfach und lesbar!** → Template Lambdas und Variadische Parameter `Args ...` wegen Tuple.



Expandiere Tuple in Funktion

C++17/20

Wir möchten folgendes:

```
invoke(std::make_tuple(1, 2.0, "Banane"),
       [](int a, double b, const std::string& c){});
```

Wie erreichen wir das:

- Template Funktion `invoke`.
- **forwarding**-Referenzen brauchen, damit `invoke(std::make_tuple(...))` möglich.
- Meta-Programming möglichst **einfach und lesbar!** → Template Lambdas und Variadische Parameter `Args ...` wegen Tuple.
- `std::get<I>(tuple)` um den `I`-ten Wert des Tuples zurückzugeben.



Expandiere Tuple in Funktion

C++17/20, [Live]

```
template<typename T, typename F>
void invoke(T&& tuple, F&& func)
{
    auto makeRange = []<typename ... Args> (std::tuple<Args ... >)
    {   //     std::index_sequence<0,1,2,3, ..., N-1>
        return std::make_index_sequence<sizeof ... (Args)>{};
    };

    auto call = [&]<std::size_t ... I> (std::index_sequence<I ... >){
        func(std::get<I>(tuple) ... );
    };

    call(makeRange(tuple));
}
```

- C++17: Lambdas sind implizit `constexpr` falls möglich.



Expandiere Tuple in Funktion

C++17/20, [Live]

```
template<typename T, typename F>
void invoke(T&& tuple, F&& func)
{
    auto makeRange = []<typename ... Args> (std::tuple<Args ... >)
    {   //     std::index_sequence<0,1,2,3, ..., N-1>
        return std::make_index_sequence<sizeof ... (Args)>{};
    };

    auto call = [&]<std::size_t ... I> (std::index_sequence<I ... >){
        func(std::get<I>(tuple) ... );
    };

    call(makeRange(tuple));
}
```

- C++17: Lambdas sind implizit `constexpr` falls möglich.



Expandiere Tuple in Funktion

C++17/20, [Live]

```
template<typename T, typename F>
void invoke(T&& tuple, F&& func)
{
    auto makeRange = []<typename ... Args> (std::tuple<Args ... >)
    {   //     std::index_sequence<0,1,2,3, ..., N-1>
        return std::make_index_sequence<sizeof ... (Args)>{};
    };

    auto call = [&]<std::size_t ... I> (std::index_sequence<I ... >){
        func(std::get<I>(tuple) ... );
    };

    call(makeRange(tuple));
}
```

- C++17: Lambdas sind implizit `constexpr` falls möglich.



Questions



Maybe Teil 3: Spezialisierung, Sfinae, etc... ?