rvalue references

```
X make_x()        // build an X with some data
{ return X(); }

int main()
{
    X x1;
    X x2(x1);                // copy
    X x3(std::move(x1));     // move: x1 no longer has any data

    x1=x2;              // copy assign
    x1=make_x();        // return value is an rvalue, so move rather than copy
}
```

rvalue references

Platzierung:

```
class X  { public: X(U); ... };

std::vector<X> v; U u;

v.push_back(X(u)); // C++98: copy X, ggf. RVO (return value optimization)

v.push_back(X(u)); // C++11: move, falls X moveable, sonst copy (ggf. RVO)

// +
v.emplace_back(u); // no copy, no move: create in place
```

```
<vector>, <list>, <deque> ....
    void push_back(const T& x);
    void push_back(T&& x);
    template <class... Args> void emplace_back(Args&&... args);
    template <class... Args> iterator emplace(const_iterator position, Args&&... args);
    iterator insert(const_iterator position, const T& x);
    iterator insert(const_iterator position, T&& x);
```

rvalue references

Perfect Forwarding Args&&... args ???

in Template-Code Parametertypen im Original verwenden:

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg arg) // not perfect: extra copy
{
  return shared_ptr<T>(new T(arg));
}

// better:
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg) // no copy, but not for rvalues
{
  return shared_ptr<T>(new T(arg));
}
... factory<X>(makeX()); // error
... factory<X>(42);      // error, despite of X(int)
```

rvalue references

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg& arg) // no copy, for lvalues
{
  return shared_ptr<T>(new T(arg));
}
//+
template<typename T, typename Arg>
shared_ptr<T> factory(const Arg& arg) // no copy, for rvalues
{
  return shared_ptr<T>(new T(arg));
}

... factory<X>(makeX()); // ok
... factory<X>(42);      // ok
```

leider auch nicht perfekt:

- 2^n Überlandungen bei n Parametern :-(
- keine move-Semantic möglich :-(

rvalue references

die Lösung: *rvalue* Referenzen selbst

bislang (C++98) war es nicht möglich Referenzen auf Referenzen zu setzen: ~~A& &~~

nun (C++11) gelten die folgenden *reference collapsing rules*:

- A& & ---> A&
- A& && ---> A&
- A&& & ---> A&
- A&& && ---> A&&

1. When called on an lvalue of type `Arg`, then `Arg` resolves to `Arg&` and hence, by the reference collapsing rules above, the argument type effectively becomes `Arg&`.
2. When called on an rvalue of type `Arg`, then `Arg` resolves to `Arg`, and hence the argument type becomes `Arg&&`.

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg) {
   return shared_ptr<T>(new T(std::forward<Arg>(arg)));
}
```

rvalue references

`std::forward ?`

bei der Übergabe von Arg den T-Konstruktor den ermittelten Typ bewahren:

dazu wird benötigt (implizit aus `<type_traits>`)

```
template< class T > struct remove_reference;     // general pattern
//template specializations:
template< class T > struct remove_reference      {typedef T type;};
template< class T > struct remove_reference<T&>  {typedef T type;};
template< class T > struct remove_reference<T&&> {typedef T type;};

und:

template<class S>
S&& forward(typename remove_reference<S>::type& a) noexcept
{
  return static_cast<S&&>(a);
}
```

rvalue references

alles zusammen:
wenn das Argument ein *lvalue* ist

```
X x; // alles was einen Namen hat ist lvalue
factory<A>(x); // Arg ---> X&

einsetzen ergibt:

shared_ptr<A> factory(X& && arg)
{
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& && forward(remove_reference<X&>::type& a) noexcept
{
  return static_cast<X& &&>(a);
}
```

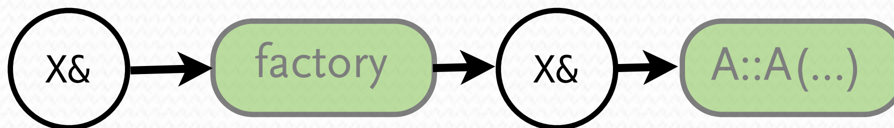
rvalue references

wenn das Argument ein *lvalue* ist

```
X x; // alles was einen Namen hat ist lvalue
factory<A>(x); // Arg ---> X&
```

```
einsetzen ergibt:

shared_ptr<A> factory(X& arg)
{
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& forward(X& a) noexcept
{
  return static_cast<X&>(a);
}
```



rvalue references

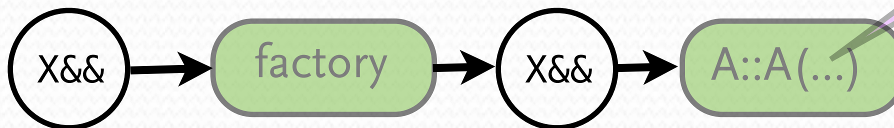
wenn das Argument ein *rvalue* ist

```
X foo(); // rvalue
factory<A>(foo()); // Arg ---> X

einsetzen ergibt:

shared_ptr<A> factory(X&& arg)
{
  return shared_ptr<A>(new A(std::forward<X>(arg)));
}
X&& forward(X& a) noexcept
{
  return static_cast<X&&>(a);
}
```

moving into A !!



rvalue references

Was ist mit mehr als einem Parameter?
auch *parameter packs* (s. *variadic templates*) können als *rvalue* Referenzen übergeben werden

Beispiel:

```
template< class... Types >
tuple<VTypes...> make_tuple( Types&&... args ); // Vtypes: „Value types“
```

Creates a tuple object, deducing the target type from the types of arguments. The deduced types are std::decay<Ti>::type (transformed as if passed to a function by value) unless application of std::decay results in std::reference_wrapper<X> for some type X, in which case the deduced type is is X&.

```
template< class T >  // more type magic
struct decay {
    typedef typename std::remove_reference<T>::type U;
    typedef typename std::conditional<
        /*if*/ std::is_array<U>::value,
        /*then*/ typename std::remove_extent<U>::type*,
        /*else*/ typename std::conditional<
            /*if*/ std::is_function<U>::value,
            /*then*/ typename std::add_pointer<U>::type,
            /*else*/ typename std::remove_cv<U>::type
    >::type>::type type;
};
```

```
template<bool B, class T, class F>
struct conditional { typedef T type; };

template<class T, class F>
struct conditional<false, T, F> { typedef F type; };
```

rvalue references

Damit funktioniert alles „wie erwartet“:

```
#include <iostream>
#include <tuple>
#include <functional>

int main()
{
    auto t1 = std::make_tuple(10, "Test", 3.14);
    std::cout << "The value of t1 is "
              << "(" << std::get<0>(t1) << ", " << std::get<1>(t1)
              << ", " << std::get<2>(t1) << ")\n";

    int n = 1;
    auto t2 = std::make_tuple(std::ref(n), n); //std::reference_wrapper
    n = 7;
    std::cout << "The value of t2 is "
              << "(" << std::get<0>(t2) << ", " << std::get<1>(t2) << ")\n";
}
```

```
The value of t1 is (10, Test, 3.14)
The value of t2 is (7, 1)
```

rvalue references

was macht `std::move` nun wirklich?

The `move` function really does very little work. All `move` does is accept either an lvalue or rvalue argument, and return it as an rvalue *without* triggering a copy construction:

```
template<class T>
typename remove_reference<T>::type&&
std::move(T&& a) noexcept
{
  typedef typename remove_reference<T>::type&& RvalRef;
  return static_cast<RvalRef>(a);
}
```

rvalue references

```
// stealing from named objects:
class X {
    char* state;
public:
    X(const char* s): state(strdup(s)) {}
    X(X&& src) {  state = src.state;
                  src.state = nullptr; }
    X(const X& src): state(strdup(src.state)) {}
    friend std::ostream& operator<<(std::ostream& out, X& x) {
        if (x.state == nullptr)
            return out<<"nullptr"<<std::endl;
        return out<<x.state<<std::endl;
    }
    ~X() { free(state); }
};

int main (){ X x1("eins"); X x2(x1); X x3(std::move(x1));
             std::cout<<x1<<x2<<x3;
}
```

DEMO: perfect1

```
nullptr
eins
eins
```

PODs (generalized)

C++98 beschränkt POD (plain old data) auf C-structs, obwohl vieles aus Klassen am Layout nichts ändert.

```
struct S { int a; }; // S is a POD
struct SS { int a; SS(int aa) : a(aa) { } };
// SS is not a (C++98-)POD
struct SSS { virtual void f(); /* ... */ };
```

In C++11, S und SS sind "standard layout types" (a.k.a. POD). SSS ist auch kein POD wegen dem vptr.

PODs sind trivial kopierbare Typen, triviale Typen, und standard-layout Typen. POD ist rekursiv definiert: Wenn alle Member POD's sind und die Klassen keine virt. Funktionen, virtual Bases oder Referenzen enthält, ist sie selbst POD

raw string literals

Backslash-Escape-Horror (insb. bei regulären Ausdrücken: Muster Wort”\”Wort)

```
string s = "\\w\\\\\\w"; // I hope I got that right
```

In einem "raw string literal" ist der backslash einfach ein backslash

```
string s = R"[\w\\\w]"; // I'm pretty sure I got that right
```

Warum nicht R”....” ? Häufig braucht man “ im quoted string!

```
R"["quoted string"]" // the string is "quoted string"
```

Was ist mit **]** in einem a raw string? Eher selten und "**[...]**" ist nur das *default delimiter pair*.
Beliebige Zeichen vor **[und** nach **]** sind erlaubt.

```
R"***["quoted string containing the usual terminator ("])"]***"
// the string is "quoted string containing the usual terminator ("])"
```

user-defined literals

Nutzerdefinierte Typen und built-in Typen sind gleichberechtigt, außer dass es keine Literale dieser Typen gibt ☹

```
"Hi!"_s // string, not ``zero-terminated array of char''
1.2_i // imaginary
123.4567891234_df // decimal floating point (IBM)
101010111000101_b // binary
123_s // seconds
123.56_km // not miles! (units)
1234567890123456789012345678901234567890_x // extended-precision
```

In C++11 sind *user-defined literals* möglich durch Definition sog. *literal operators*.

```
constexpr complex<double> operator "" _i(long double d)
// imaginary literal
{ return {0,d}; } // complex is a literal type

std::string operator"" _s (const char* p, size_t n) // std::string literal
{ return string(p,n); } // requires free store allocation
```

constexpr erlaubt statische Auswertung z.B. `1+2_i`

user-defined literals

Argumente können “cooked” – als Literal des Typs (nur Zahlen!) ohne Suffix oder aber “uncooked” – als String übergeben werden.

Es gibt 4 Arten von user-defined literals:

- integer literal: Literaloperator mit einem Argument vom Typ **unsigned long long** oder **const char*** .
- floating-point literal: Literaloperator mit einem Argument vom Typ **long double** oder **const char*** .
- string literal: Literaloperator mit zwei Argumenten der Typen **(const char*, size_t)** ! 2. Argument darf nicht fehlen: Wenn man eine ‘andere Sorte Strings’ haben will, muss die Länge immer bekannt sein!
- character literal: Literaloperator mit einem Argument vom Typ **char** .

user-defined literals

```
Bignum operator"" _x(const char* p) { return Bignum(p); }
void f(Bignum);
f(1234567890123456789012345678901234567890_x);
// C-style string "1234567890123456789012345678901234567890"
// is passed to operator"" _x().
```

Suffixe sind (gewollt) kurz – Namespaces vermeiden *name clashes*:

```
namespace Numerics { // ...
 class Bignum { /* ... */ };
 namespace literals { Bignum operator"" _x(char const*); }
}
using namespace Numerics::literals;
```

user-defined literals

1

```
#include <iostream>
#include <complex>

std::complex<long double> operator "" _i(long double d) // cooked form
{
    return std::complex<long double>(0, d);
}

auto val = 3.14_i; // val = complex<long double>(0, 3.14)

int main() {

    std::cout << val;       // --> (0,3.14)
    return 0;
}

// compiles with g++ 4.7, NOT Clang3.0, NOT MS VS C++11
```

user-defined literals

```
// works with variadic templates too :-)
namespace literals{
 template <char C> int bin();
 template <>       int bin<'1'>() { return 1; }
 template <>       int bin<'0'>() { return 0; }
 template <char C, char D, char... ES>
 int bin() {
        return bin<C>() << (sizeof...(ES)+1) | bin<D, ES...>();
 }
 template <char... CS> int operator "" _bin() {
        return bin<CS...>();
 }
}

int main() {
    using namespace literals;
    int eins = 1_bin;
    int f = 1111_bin;
}                          // std::cout<<f; ???
```

operator “ “ _...

kann / darf nicht / muss