Back to Basics:

Move Semantics

(part 2 of 2)

Klaus Iglberger, CppCon 2019

C++ Trainer since 2016

Senior Software Engineer at Siemens

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Regular presenter at C++ conferences

Klaus Iglberger

Content

Back to Basics: Move Semantics (Part 1)

- The Basics of Move Semantics
- The New Special Member Functions
  - The Move Constructor
  - The Move Assignment Operator
- Parameter Conventions

Back to Basics: Move Semantics (Part 2)

- Forwarding References
  - Perfect Forwarding
  - The Perils of Forwarding References
  - Overloading with Forwarding References
- Move Semantics Pitfalls

Forwarding References

This could be it, life would be great, but …

Forwarding References

```
template< typename T >
void f( T&& x );          // Forwarding reference

auto&& var2 = var1;       // Forwarding reference
```

Forwarding references represent …

- … an lvalue reference if they are initialized by an lvalue;
- … an rvalue reference if they are initialized by an rvalue.

Rvalue references are forwarding references if they …

- … involve type deduction;
- … appear in exactly the form `T&&` or `auto&&`.

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );
}
```

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );
}
```

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );
}
```

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );    // Prints 'foo(T&&)'
}
```

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );   // Prints 'foo(T&&)'
}
```

Forwarding References

```
template< >
void foo( Widget& && ) {
    puts( ”foo(T&&)” );
}

int main()
{
    Widget w{};
    foo( w );   // Prints 'foo(T&&)'
}
```

Forwarding References

```
template< >
void foo( Widget& && ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );   // Prints 'foo(T&&)'
}
```

Forwarding References

```
template< >
void foo( Widget& && ) {
    puts( ”foo(T&&)” );
}

int main()
{
    Widget w{};
    foo( w );   // Prints 'foo(T&&)'
}
```

Reference Collapsing

& &	➜	&
&& &	➜	&
& &&	➜	&
&& &&	➜	&&

Forwarding References

```
template< >
void foo( Widget& && ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );   // Prints 'foo(T&&)'
}
```

Reference Collapsing

& &	→	&
&& &	→	&
& &&	→	&
&& &&	→	&&

Forwarding References

```
template< >
void foo( Widget& ) {
   puts( ”foo(T&&)” );
}

int main()
{
   Widget w{};
   foo( w );   // Prints 'foo(T&&)'
}
```

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
  foo( Widget{} );
}
```

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
  foo( Widget{} );   // Prints foo(T&&)
}
```

Forwarding References

```
template< typename T >
void foo( T&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
  foo( Widget{} );   // Prints foo(T&&)
}
```

Forwarding References

```
template< >
void foo( Widget&& ) {
   puts( ”foo(T&&)” );
}

int main()
{
  foo( Widget{} );   // Prints foo(T&&)
}
```

Perfect Forwarding

Perfect Forwarding

```
namespace std {

template<typename T, ???>
unique_ptr<T> make_unique(???)
{
    return unique_ptr<T>(new T(???));
}

} // namespace std
```

How shall we pass arguments to the `make_unique()` function, which forwards these arguments to the constructor of T?

Perfect Forwarding

```
namespace std {

template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg arg)
{
    return unique_ptr<T>(new T(arg));
}

} // namespace std


std::make_unique<int>( 1 );     // Cheap extra copy

std::make_unique<Widget>( w );  // Expensive extra copy
```

Perfect Forwarding

```
namespace std {

template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg& arg)
{
    return unique_ptr<T>(new T(arg));
}

} // namespace std


std::make_unique<int>( 1 );  // Compilation error, rvalue
```

Perfect Forwarding

```
namespace std {

template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg const& arg)
{
    return unique_ptr<T>(new T(arg));
}

} // namespace std


struct Example { Example( int& ); };

int i{ 1 };
std::make_unique<Example>( i );  // Always adds const
```

Perfect Forwarding

```
namespace std {

template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg&& arg)
{
    return unique_ptr<T>(new T(arg));
}

} // namespace std
```

Solution: Pass-by-forwarding reference!

Perfect Forwarding

```
namespace std {

template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg&& arg)
{
    return unique_ptr<T>(new T(arg));
}

} // namespace std
```

lvalue!

Solution: Pass-by-forwarding reference!

std::forward

- std::forward conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- std::forward does not forward anything

```
template< typename T >
T&& forward( std::remove_reference_t<T>& t ) noexcept
{
   return static_cast<T&&>( t );
}
```

Perfect Forwarding

```
namespace std {

template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg&& arg)
{
    return unique_ptr<T>(new T(std::forward<Arg>(arg)));
}

} // namespace std
```

Solution: Pass-by-forwarding reference!

Perfect Forwarding

```
namespace std {

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

} // namespace std
```

Final solution, extended to take an arbitrary number of parameters.

The Mechanics of std::forward

The Mechanics of std::forward

```
namespace std {

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

} // namespace std
```

The Mechanics of `std::forward`

- `std::forward` conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- `std::forward` does not forward anything

```
template< typename T >
T&& forward( std::remove_reference_t<T>& t ) noexcept
{
   return static_cast<T&&>( t );
}
```

The Mechanics of `std::forward`

- `std::forward` conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- `std::forward` does not forward anything

```
template< >
Widget& && forward( std::remove_reference_t<Widget&>& t ) noexcept
{
   return static_cast<Widget& &&>( t );
}
```

The Mechanics of std::forward

- std::forward conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- std::forward does not forward anything

reference collapsing!

```
template< >
Widget& && forward( std::remove_reference_t<Widget&>& t ) noexcept
{
   return static_cast<Widget& &&>( t );
}
```

The Mechanics of `std::forward`

- `std::forward` conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- `std::forward` does not forward anything

```
template< >
Widget& forward( std::remove_reference_t<Widget&>& t ) noexcept
{
   return static_cast<Widget&>( t );
}
```

The Mechanics of `std::forward`

- `std::forward` conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- `std::forward` does not forward anything

```
template< >
Widget& forward( Widget& t ) noexcept
{
   return static_cast<Widget&>( t );
}
```

The Mechanics of `std::forward`

- `std::forward` conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- `std::forward` does not forward anything

```
template< >
Widget&& forward( std::remove_reference_t<Widget>& t ) noexcept
{
   return static_cast<Widget&&>( t );
}
```

The Mechanics of std::forward

- std::forward conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- std::forward does not forward anything

```
template< >
Widget&& forward( Widget& t ) noexcept
{
   return static_cast<Widget&&>( t );
}
```

The Mechanics of `std::forward`

- `std::forward` conditionally casts its input into an rvalue reference
  - If the given value is an lvalue, cast to an lvalue reference
  - If the given value is an rvalue, cast to an rvalue reference
- `std::forward` does not forward anything

```
template< typename T >
T&& forward( std::remove_reference_t<T>& t ) noexcept
{
   return static_cast<T&&>( t );
}
```

The Mechanics of `std::forward`

- `std::move` unconditionally casts its input into an rvalue reference
- `std::move` does not move anything

```
template< typename T >
std::remove_reference_t<T>&&
   move( T&& t ) noexcept
{
   return static_cast<std::remove_reference_t<T>&&>( t );
}
```

The Perils of Forwarding References

The Perils of Forwarding References

```
struct Person {
   Person( const std::string& name );      // (1)
   template< typename T > Person( T&& );   // (2)
};
```

The Perils of Forwarding References

```
struct Person {
   Person( const std::string& name );     // (1)
   template< typename T > Person( T&& );  // (2)
};

int main()
{
   Person p1( "Bjarne" );      // calls ctor (2);
                               // argument type is char[7]




}
```

The Perils of Forwarding References

```
struct Person {
   Person( const std::string& name );     // (1)
   template< typename T > Person( T&& );  // (2)
};

int main()
{
   Person p1( "Bjarne" );         // calls ctor (2);
                                  // argument type is char[7]
   std::string name( "Herb" );
   Person p2( name );             // calls ctor (2);
                                  // argument type is NOT const


}
```

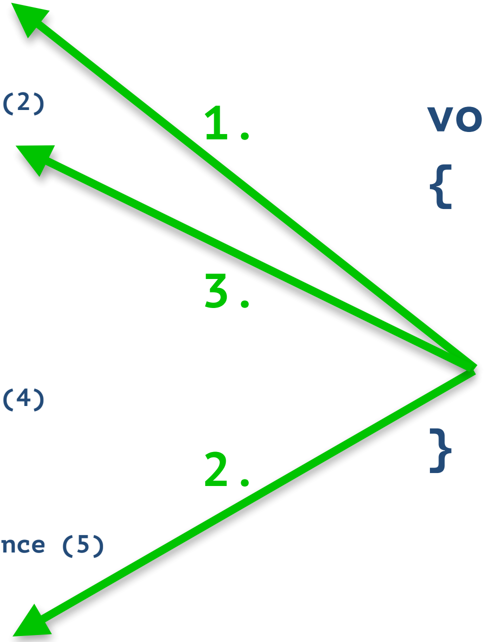
The Perils of Forwarding References

```
struct Person {
   Person( const std::string& name );     // (1)
   template< typename T > Person( T&& );  // (2)
};

int main()
{
   Person p1( "Bjarne" );          // calls ctor (2);
                                   // argument type is char[7]
   std::string name( "Herb" );
   Person p2( name );              // calls ctor (2);
                                   // argument type is NOT const

   Person p3( p1 );                // calls ctor (2), not copy ctor;
}                                  // argument type is NOT const
```

Overloading with Forwarding References

Overloading with Forwarding References

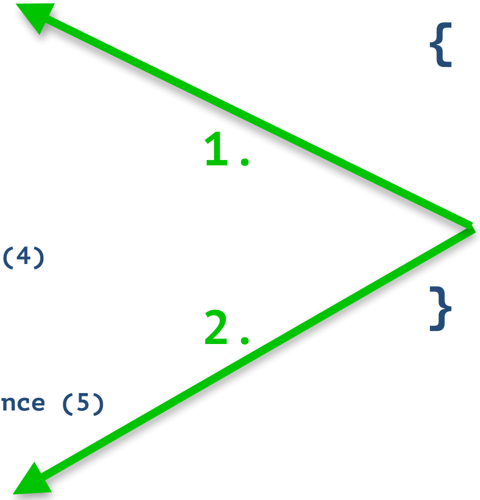
```
// Function with lvalue reference (1)
void f( Widget& );

// Function with lvalue reference-to-const (2)
void f( const Widget& );

// Function with rvalue reference (3)
void f( Widget&& );

// Function with rvalue reference-to-const (4)
void f( const Widget&& );

// Function template with forwarding reference (5)
template< typename T >
void f( T&& );

// Function template with rvalue reference-to-const (6)
template< typename T >
void f( const T&& );
```



```
void g()
{
   Widget w{};

   f( w );
}
```

Overloading with Forwarding References

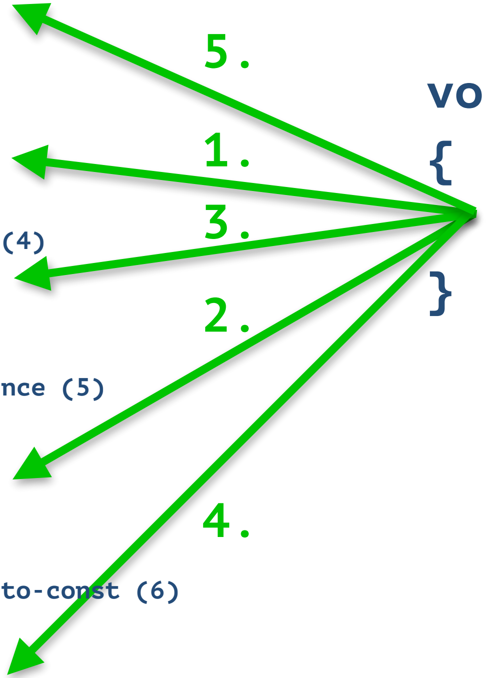
```
// Function with lvalue reference (1)
void f( Widget& );

// Function with lvalue reference-to-const (2)
void f( const Widget& );

// Function with rvalue reference (3)
void f( Widget&& );

// Function with rvalue reference-to-const (4)
void f( const Widget&& );

// Function template with forwarding reference (5)
template< typename T >
void f( T&& );

// Function template with rvalue reference-to-const (6)
template< typename T >
void f( const T&& );
```



```
void g()
{
   const Widget w{};

   f( w );
}
```

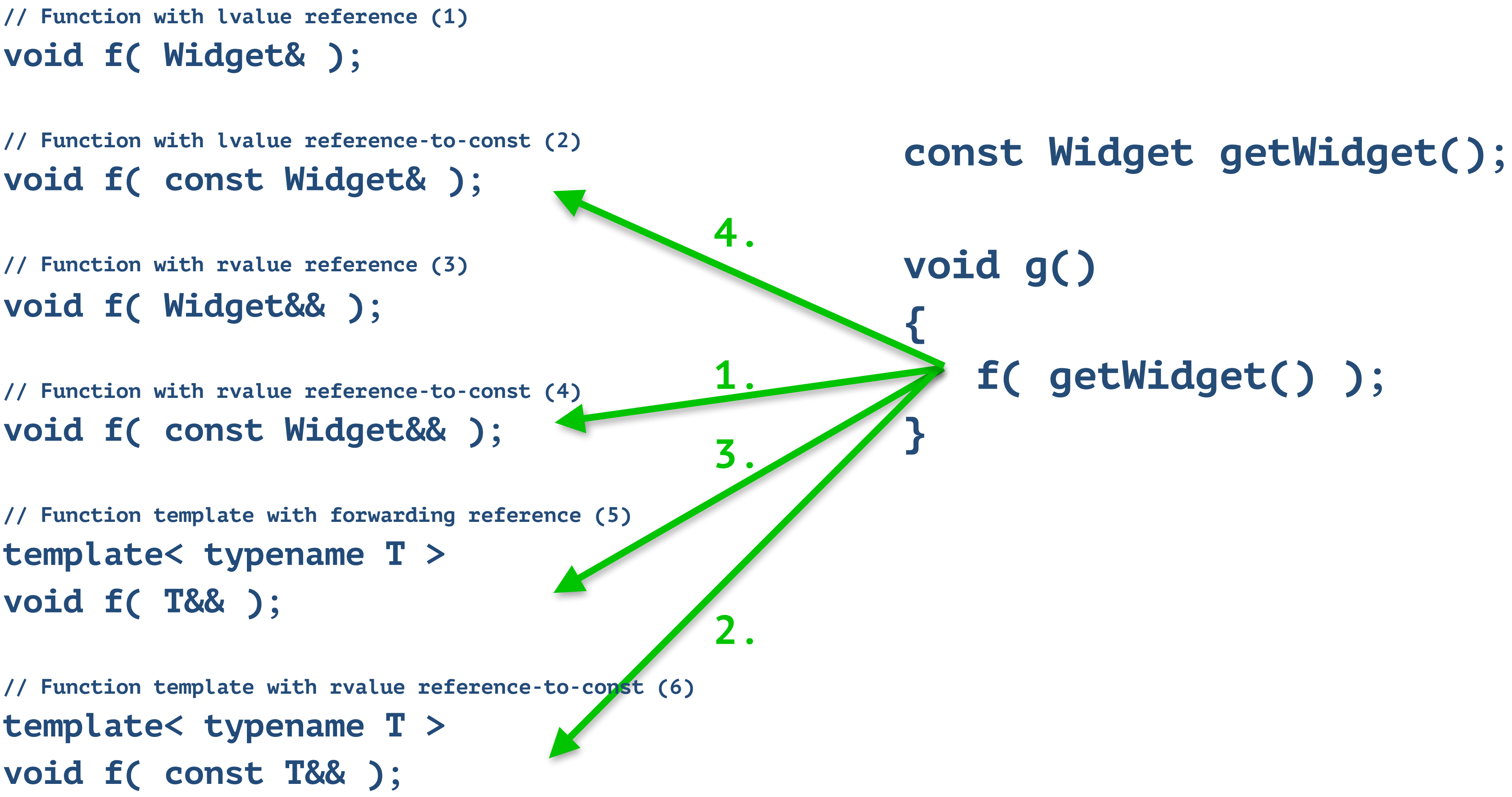
Overloading with Forwarding References

```
// Function with lvalue reference (1)
void f( Widget& );

// Function with lvalue reference-to-const (2)
void f( const Widget& );

// Function with rvalue reference (3)
void f( Widget&& );

// Function with rvalue reference-to-const (4)
void f( const Widget&& );

// Function template with forwarding reference (5)
template< typename T >
void f( T&& );

// Function template with rvalue reference-to-const (6)
template< typename T >
void f( const T&& );
```



```
Widget getWidget();

void g()
{
   f( getWidget() );
}
```

Overloading with Forwarding References



```
// Function with lvalue reference (1)
void f( Widget& );

// Function with lvalue reference-to-const (2)
void f( const Widget& );

// Function with rvalue reference (3)
void f( Widget&& );

// Function with rvalue reference-to-const (4)
void f( const Widget&& );

// Function template with forwarding reference (5)
template< typename T >
void f( T&& );

// Function template with rvalue reference-to-const (6)
template< typename T >
void f( const T&& );
```

```
const Widget getWidget();

void g()
{
   f( getWidget() );
}
```

Overloading with Forwarding References

Effective Modern C++, Item 26: Avoid overloading on universal references (Scott Meyers)

Move Semantics Pitfalls

R-VALUE REFERENCE EXPERT

Move Semantics Pitfalls (Example 1)

```
class A {
 public:
   template< typename T >
   A( T&& t )
     : b_( std::move( t ) )
   {}
 private:
   B b_;
};
```

Move Semantics Pitfalls (Example 1)

```
class A {
 public:
   template< typename T >
   A( T&& t )
     : b_( std::move( t ) )
   {}
 private:
   B b_;
};
```

Move Semantics Pitfalls (Example 1)

```
class A {
 public:
   template< typename T >
   A( T&& t )
     : b_( std::forward<T>( t ) )
   {}
 private:
   B b_;
};
```

Move Semantics Pitfalls (Example 2)

```
template< typename T >
class A {
 public:
   A( T&& t )
     : b_( std::forward<T>( t ) )
   {}
 private:
   B b_;
};
```

● ● ● ● ● ● ● ● ● ● ● ● ● ● ●

Move Semantics Pitfalls (Example 2)

```
template< typename T >
class A {
 public:
   A( T&& t )
     : b_( std::forward<T>( t ) )
   {}
 private:
   B b_;
};
```

class (!) template parameter

rvalue reference!

Move Semantics Pitfalls (Example 2)

```
template< typename T >
class A {
 public:
   A( T&& t )
     : b_( std::forward<T>( t ) )
   {}
 private:
   B b_;
};
```

class (!) template parameter

rvalue reference!

Move Semantics Pitfalls (Example 2)

```
template< typename T >
class A {
 public:
   A( T&& t )
     : b_( std::move( t ) )
   {}
 private:
   B b_;
};
```

Move Semantics Pitfalls (Example 3)

```
class A {
 public:
   template< typename T >
   A( T&& t )
     : b_( std::forward<T>( t ) )
     , c_( std::forward<T>( t ) )
   {}
 private:
   B b_;
   C c_;
};
```

Move Semantics Pitfalls (Example 3)

```
class A {
 public:
   template< typename T >
   A( T&& t )
     : b_( std::forward<T>( t ) )
     , c_( std::forward<T>( t ) )
   {}
 private:
   B b_;
   C c_;
};
```

Move Semantics Pitfalls (Example 3)

```
class A {
 public:
   template< typename T >
   A( T&& t )
     : b_( t )
     , c_( std::forward<T>( t ) )
   {}
 private:
   B b_;
   C c_;
};
```

Move Semantics Pitfalls (Example 4)

```
class A {
 public:
   template< typename T1, typename T2 >
   A( T1&& t1, T2&& t2 )
     : b_( std::forward<T1>( t1 ) )
     , c_( std::forward<T2>( t2 ) )
   {}
 private:
   B b_;
   C c_;
};
```

● ● ● ● ● ● ● ● ● ● ● ● ● ● ●

Move Semantics Pitfalls (Example 4)

What if these are references to the same object?

```
class A {
 public:
   template< typename T1, typename T2 >
   A( T1&& t1, T2&& t2 )
     : b_( std::forward<T1>( t1 ) )
     , c_( std::forward<T2>( t2 ) )
   {}
 private:
   B b_;
   C c_;
};
```

Move Semantics Pitfalls (Example 4)

```
class A {
 public:
   template< typename T1, typename T2 >
   A( T1&& t1, T2&& t2 )
     : b_( std::forward<T1>( t1 ) )
     , c_( std::forward<T2>( t2 ) )
   {}
 private:
   B b_;
   C c_;
};
```

✅

Move Semantics Pitfalls (Example 5)

```
template< typename... Args >
std::unique_ptr<Widget> create( Args&&... args )
{
   auto uptr( std::make_unique<Widget>(
      std::forward<Args>(args)... ) );
   return std::move( uptr );
}
```

Move Semantics Pitfalls (Example 5)

```
template< typename... Args >
std::unique_ptr<Widget> create( Args&&... args )
{
   auto uptr( std::make_unique<Widget>(
      std::forward<Args>(args)... ) );
   return std::move( uptr );  // Prevents RVO
}
```

Move Semantics Pitfalls (Example 5)

```
template< typename... Args >
std::unique_ptr<Widget> create( Args&&... args )
{
   auto uptr( std::make_unique<Widget>(
      std::forward<Args>(args)... ) );
   return std::move( uptr );  // Prevents RVO
}
```

Move Semantics Pitfalls (Example 5)

```
template< typename... Args >
std::unique_ptr<Widget> create( Args&&... args )
{
   return std::make_unique<Widget>(
      std::forward<Args>(args)... );

}
```

Move Semantics Pitfalls (Example 5)

```
template< typename... Args >
std::unique_ptr<Widget>&& create( Args&&... args )
{
   return std::make_unique<Widget>(
      std::forward<Args>(args)... );

}
```

Returns reference to local object!

Move Semantics Pitfalls (Example 5)

Core Guideline F.45: Don’t return a `T&&`

Move Semantics Pitfalls (Example 6)

```
template< typename T >
void foo( T&& )
{

   if constexpr( std::is_integral_v<T> )
   {
      // Deal with integral types
   }
   else
   {
      // Deal with non-integral types
   }
}
```

Move Semantics Pitfalls (Example 6)

```
template< typename T >
void foo( T&& )
{

   if constexpr( std::is_integral_v<T> )
   {
      // Deal with integral types
   }
   else
   {
      // Deal with non-integral types
   }
}
```

for lvalues this is a reference!

Move Semantics Pitfalls (Example 6)

```
template< typename T >
void foo( T&& )
{
   using NoRef = std::remove_reference_t<T>;
   if constexpr( std::is_integral_v<NoRef> )
   {
      // Deal with integral types
   }
   else
   {
      // Deal with non-integral types
   }
}
```

That's you, the rvalue reference expert!

klaus.iglberger@gmx.de