Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 3 – Type Deduction

Thomas Corbat / Felix Morgner
Rapperswil, 08.03.2021
FS2021

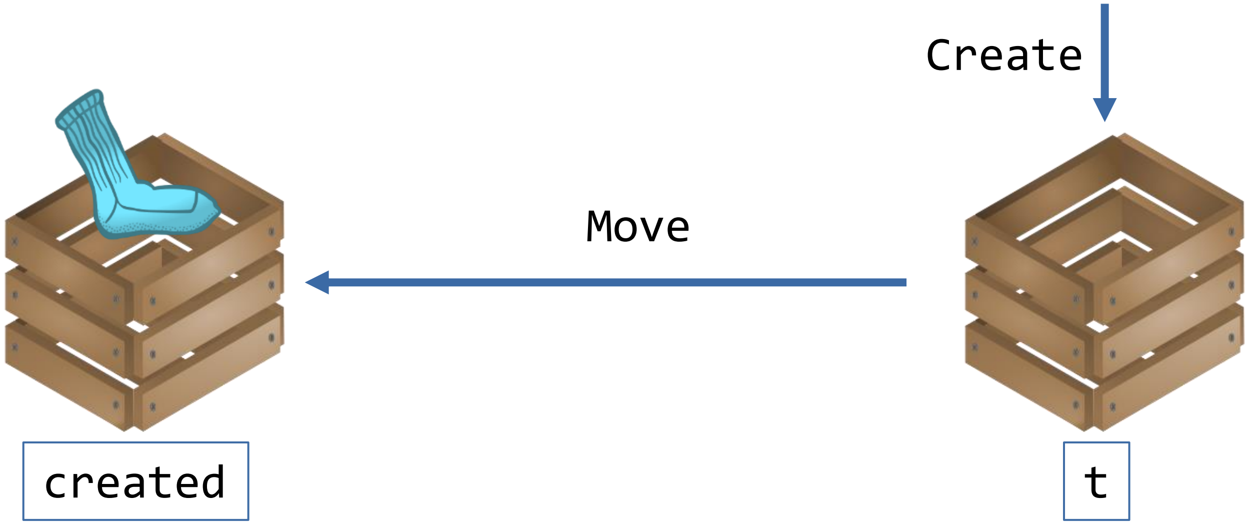
- **Topics:**
  - Recap: Move Semantics
  - Type Deduction and Forwarding References
  - `auto` and `decltype` Keywords
  - Perfect Forwarding

Goals

- **You recognize forwarding references and can decide what they become**
- **You can determine the deduced type for function templates and `auto`/`decltype(auto)`**
- **You can design function template signatures that adapt to lvalues and rvalues efficiently even for multiple parameters**
- **You can explain and apply perfect forwarding**

Recap: Move Semantics

Moving Elements




```
Create Thing
Move Thing
```

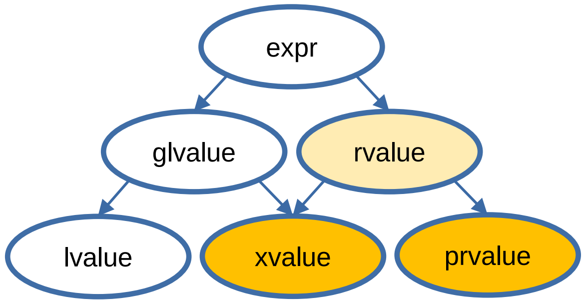
```cpp
struct MoveableThing {
  MoveableThing() {
    std::cout << "Create Thing\n";
  }

  MoveableThing(MoveableThing &&) {
    std::cout << "Move Thing\n";
  }
};

MoveableThing create() {
  MoveableThing t{};
  return t;
}

int main() {
  MoveableThing created = create();
}
```

- **Compile in GCC with `-fno_elide_constructors`**
- **Pre C++17: One additional move would happen without optimization**

Example Move Constructor

```
struct ContainerForBigObject {
  ContainerForBigObject()
      : resource{std::make_unique<BigObject>()} {}

  ContainerForBigObject(ContainerForBigObject const & other)
      : resource{std::make_unique<BigObject>(*other.resource)} {}

  ContainerForBigObject(ContainerForBigObject && other)
      : resource{std::move(other.resource)} {}

  ContainerForBigObject & operator=(ContainerForBigObject const & other) {
    resource = std::make_unique<BigObject>(*other.resource);
    return *this;
  }

  ContainerForBigObject & operator=(ContainerForBigObject && other) {
    std::swap(resource, other.resource);
    //resource = std::move(other.resource) is possible too
    return *this;
  }

private:
  std::unique_ptr<BigObject> resource;
};
```

- **References for rvalues**
  - Syntax: `<Type> &&`
  - Binds to an rvalue (xvalue or prvalue)

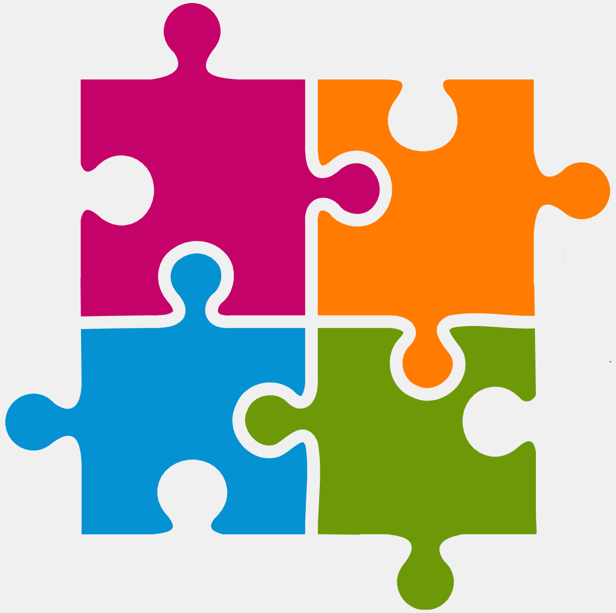


- **Argument is either a literal or a temporary object**

```
std::string createGlass();

void fancy_name_for_function() {
  std::string mug{"cup of coffee"};
  std::string && glass_ref = createGlass(); //life-extension of temporary
  std::string && mug_ref = std::move(mug);  //explicit conversion lvalue to xvalue
  int && i_ref = 5;                         //binding rvalue reference to prvalue
}
```

- **Beware: Parameters and variables declared as rvalue references are lvalues in the context of function bodies! (Everything with a name is an lvalue)**
- **Beware 2.0: `T&&/auto&&` is not always an rvalue reference! (We'll come to that today)**

Type deduction



Based on Modern Effective C++ by Scott Meyers

Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

- **In some contexts `T&&` does not necessarily mean rvalue reference**
- **Exceptions**
  - `auto &&`
  - `T &&` when template type deduction applies for type T
- **In these cases the reference can bind to rvalues and lvalues depending on the context**

```
template<typename T>
void f(T && param);
```

```
int x = 23;
f(x);           //lvalue
```

```
void f(int & param);
```

```
f(23); //rvalue
```

```
void f(int && param);
```

```
template<typename T>
void f(ParamType param);
```

- **`T` and `ParamType` are not necessarily exactly the same type**

```
template<typename T>
void f(T const & param);
```

- **Now what is `T` and `ParamType` for the following call?**

```
int x = 0;
f(x);
```

- **T: `int`**
- **ParamType: `int const &`**

- **Context:**

```
template<typename T>
void f(ParamType param);
```

- **Deduction of type `T` depends on the structure of `ParamType`**

- **Cases:**

1. ParamType is a value type (e.g. `void f(T param)`)
2. ParamType is a reference (e.g. `void f(T & param)`)
3. ParamType is a forwarding reference (exactly: `void f(T && param)`)

Note: `ParamType` might be a nested composition of templates (e.g. `void f(std::vector<T> param)`)

- **ParamType is a value type**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Ignore `const` of `<expr>` (outermost)
  3. Pattern match `<expr>`'s type against ParamType to figure out T

```
template<typename T>
void f(T param);
```

```
f(<expr>);
```

Declarations:

```
int         x   = 23;
int const   cx  = x;
int const & crx = x;
```

Calls:	Instances:	Deduced Ts:
`f(x);` →	`void f(int param);` →	`T = int`
`f(cx);` →	`void f(int param);` →	`T = int`
`f(crx);` →	`void f(int param);` →	`T = int`

Rules for Type Deduction: Value Type

- **ParamType is a value type**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Ignore `const` of `<expr>` (outermost)
  3. Pattern match `<expr>`'s type against ParamType to figure out T
- **Example const pointer to const char**

```
template<typename T>
void f(T param);
```

```
f(<expr>);
```

Call:

```
char const * const ptr = "...";
f(ptr);
```

Instance:

```
void f(char const * param);
```

Deduced `T`:

```
T = char const *
```

- **Note: If ParamType is a pointer type, the same rules apply as for value types. Except hat the pointer is pattern-matched and not contained in the deduced type.**

Rules for Type Deduction: Reference Type

- **ParamType is a reference type, but not a forwarding reference**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Pattern match `<expr>`’s type against ParamType to figure out T

```
template<typename T>
void f(T & param);
```

```
f(<expr>);
```

- **Examples for References:**

Declarations:

```
int             x   = 23;
int const       cx  =  x;
int const &     crx =  x;
```

Calls:		Instances:		Deduced Ts:
`f(x);`	➡	`void f(int & param);`	➡	`T = int`
`f(cx);`	➡	`void f(int const & param);`	➡	`T = int const`
`f(crx);`	➡	`void f(int const & param);`	➡	`T = int const`

Rules for Type Deduction: Reference Type (const)

- **ParamType is a reference type, but not a forwarding reference**
- **Steps:**
  1. `<expr>` is a reference type: ignore the reference
  2. Pattern match `<expr>`'s type against ParamType to figure out T
- **Examples for Const References:**

```
template<typename T>
void f(T const & param);
```

```
f(<expr>);
```

Declarations:

```
int           x   = 23;
int const     cx  =  x;
int const &   crx =  x;
```

Calls:		Instances:		Deduced Ts:
`f(x);`	➡	`void f(int const & param);`	➡	`T = int`
`f(cx);`	➡	`void f(int const & param);`	➡	`T = int`
`f(crx);`	➡	`void f(int const & param);`	➡	`T = int`

- **ParamType is a forwarding reference**
- **Cases:**
  1. `<expr>` is an lvalue: T and ParamType become lvalue references!
  2. Otherwise (if `<expr>` is an rvalue): Rules for references apply

```
template<typename T>
void f(T && param);
```

```
f(<expr>);
```

Declarations:

```
int             x   = 23;
int const       cx  =  x;
int const &     crx =  x;
```

Calls:		Instances:		Deduced Ts:
`f(x);`	➡	`void f(int & param);`	➡	`T = int &`
`f(cx);`	➡	`void f(int const & param);`	➡	`T = int const &`
`f(crx);`	➡	`void f(int const & param);`	➡	`T = int const &`
`f(27);`	➡	`void f(int && param);`	➡	`T = int`

- **What happens if `initializer_lists` are used for template type deduction?**
  - It does not work!

```
template<typename T>
void f(T param);
```

```
f({23});        //error
```

- **Correct way:**

```
template<typename T>
void f(std::initializer_list<T> param);
```

```
f({23});        //T = int
                //ParamType = std::initializer_list<int>
```

Keywords `auto` and `decltype`

- **Essentially type deduction for `auto` is the same as we have seen before**
- **`auto` takes the place of `T`**

```
auto x = 23;                  //auto is a value type
auto const cx = x;            //auto is a value type
auto & rx = x;                //auto is a reference type

auto && uref1 = x;            //x is an lvalue, uref1 is int &
auto && uref2 = cx;           //cx is an lvalue, uref2 is int const &
auto && uref3 = 23;           //23 is an rvalue, uref3 is int &&
```

- **Special case**

```
auto init_list1 = {23};       //std::initializer_list<int>
auto init_list2{23};          //int, was std::initializer_list<int>1
auto init_list3{23, 23};      //Error, requires one single argument
```

[1]Fixed in C++17 (N3922) – Some compiler vendors have retroactively applied this fix to earlier C++ versions

`auto` Return Type Deduction

- **Since C++14 it is possible to use `auto` as return type and `auto` for parameter declarations in lambdas and functions**
  - Body must be available to deduce the type
- **Rules of these uses of `auto` follow the rules of template type deduction**

```
auto createInitList() {
  return {1, 2, 3};
}
```

```
[](auto p) {
  ...
}
```

```
auto createInt() {
  return 23;
}
```

```
void f(auto p) {
  ...
}
```

This is a GCC Extension, not a Standard C++ Feature
Will be available with C++20 Concepts

Type Deduction for `decltype`

- **`decltype` can be applied to an expression: `decltype(x)`**
  - Represents the declared type of a name expression
  - `decltype(auto)` deduces the type, but does not strip references like `auto`

```
int             x       = 23;
int const       cx      = x;
decltype(cx)    cx_too  = cx; //type of cx_too is int const
int &           rx      = x;
decltype(rx)    rx_too  = rx; //type of rx_too is int &

auto            just_x  = rx; //type of just_x is int
decltype(auto) more_rx = rx; //type of more_rx is int &
```

Type Deduction for decltype: When Declaring Return Types

- **`decltype(auto)` allows deduction of inline function return types**

```
template<typename Container, typename Index>
decltype(auto) access(Container & c, Index i) {
  return c[i];
}
```

- **`decltype` can take an expression depending on parameters for specifying trailing return types**

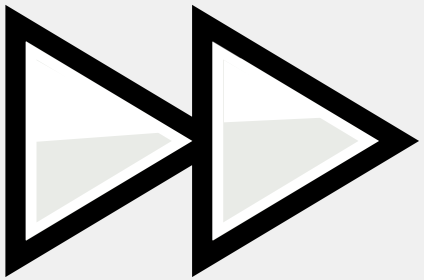
```
template<typename Container, typename Index>
auto access(Container & c, Index i) -> decltype(c[i]) {
  return c[i];
}
```

Rules for `decltype(auto)`

- **Unparenthesized variable name or data member**
  - T - Type of the expression (retains reference)
- **Expression of value category xvalue**
  - T&& - Rvalue reference to type of the expression
- **Expression of value category lvalue**
  - T& - Lvalue reference to type of the expression
- **Expression of value category prvalue**
  - T – Value type of the expression

```
decltype(auto) funcName() {
  int local = 42;
  return local; //decltype(local) => int
}
decltype(auto) funcNameRef() {
  int local = 42;
  int & lref = local;
  return lref; //decltype(lref) => int &
}
decltype(auto) funcXvalue() {
  int local = 42;
  return std::move(local); //int && -> bad
}
decltype(auto) funcLvalue() {
  int local = 42;
  return (local); //int & -> bad
}
decltype(auto) funcPrvalue() {
  return 5;  //int
}
```

Perfect Forwarding



Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

Use Case: Log Wrapper

- **Example: You have a function that does something, takes a single parameter and is overloaded for const references and rvalue references. There might be further overloads with different parameter types.**

```
void do_something(S const &);
void do_something(S &&);
```

- **Now you want to have a template that logs your operation.**

```
template<typename T>
void log_and_do(T param) {
  //log
  do_something(param);
}
```

- **This might imply a copy of `param`**

- **Let's adapt the template to use a reference to T**

```
template<typename T>
void log_and_do(T & param) {
  //log
  do_something(param);
}
```

- **Now `log_and_do` cannot be called with rvalues anymore**

```
log_and_do(23);
log_and_do(create_param());
```

- **Let's adapt the template to use a `const` reference to `T`**

```
template<typename T>
void log_and_do(T const & param) {
  //log
  do_something(param);
}
```

- **Like all versions before this prevents move semantic, as `param` is always an lvalue**
  - The overload to `do_something(ParamType &&)` will never be selected

- **Let's add an overload with an rvalue reference parameter**

```
template<typename T>
void log_and_do(T && param) {
  //log
  do_something(std::move(param));
}
```

- **That is not optimal**
  - Code duplication (only one implementation of `log_and_do` would be preferable)
  - If we have multiple parameters we had code exponentiation if we wanted to provide every combination of lvalue and rvalue parameters (2^n)
  - Overloading with forwarding references is very greedy, usually provides the best match
- **Wait! Didn't we call T && forwarding references? Don't they adapt to whatever is passed as an argument?**
  - Yes! But, `param` is always an lvalue and `std::move(param)` is always an rvalue.

- **We need something that is aware of the actual template parameter type**
- **Recap from Forwarding References: We know whether param was an lvalue or an rvalue**

```
int           x   = 23;
int const     cx  = x;
int const & crx = x;
```

```
log_and_do(x)                    -> T = int &          } lvalues
log_and_do(cx)                   -> T = int const &
log_and_do(crx)                  -> T = int const &
log_and_do(27)                   -> T = int            } rvalues
log_and_do(std::move(x))         -> T = int
```

- **If T is of reference type we need to pass an lvalue otherwise we need to pass an rvalue**
- **How can we do it?**
  - std::forward

```
template<typename T>
void log_and_do(T && param) {
  //log
  do_something(std::forward<T>(param));
}
```

Inside `std::forward`

- **What does `std::forward` do?**
  - It's a "conditional" cast to an rvalue reference...
  - This allows arguments to be treated as what they originally were (lvalue or rvalue references)
- **Implementation is similar to the following (there is also an overload for rvalue references):**

```
template<typename T>
decltype(auto) forward(std::remove_reference_t<T> & param) {
  return static_cast<T &&>(param);
}
```

- **If `T` is of value type, `T &&` is an rvalue reference in the return expression**
- **If `T` is of lvalue reference type, the resulting type is an rvalue reference to an lvalue reference**
  - Example: if `T = int &` then `T &&` would mean `int & &&`
- **What is "`<Type> & &&`" supposed to mean?**
  - The references collapse (become an lvalue reference if one is present): `<Type> &`

Example: Call with Lvalue

```
template<typename T>
void log_and_do(T && param) {
  do_something(std::forward<T>(param));
}
```

- **Let's consider a call with an lvalue**

```
Content c{};
log_and_do(c);
```

- **Instantiation of `log_and_do`:**

```
void log_and_do(Content & param) {
  do_something(std::forward<Content &>(param));
}
```

- **Instantiation of `std::forward<Content &>(param)`:**

```
decltype(auto) forward(std::remove_reference_t<Content &> & param) {
  return static_cast<Content & &&>(param);
}
```

Example: Call with Lvalue Continued


```
decltype(auto) forward(std::remove_reference_t<Content &> & param) {
  return static_cast<Content & &&>(param);
}
```

- **Parameter type applies `std::remove_reference_t` and `Content & &&` collapses to `Content &`**

```
decltype(auto) forward(Content & param) {
  return static_cast<Content &>(param);
}
```

- **As a result `std::forward<T>(param)` yields an lvalue reference to `param`**

```
template<typename T>
void log_and_do(T && param) {
  do_something(std::forward<T>(param));
}
```

- **Eventually `do_something(S const &)` will be called (lvalue overload)**

Example: Call with Rvalue

```
template<typename T>
void log_and_do(T && param) {
  do_something(std::forward<T>(param));
}
```

- **Let's consider a call with an rvalue**

```
log_and_do(Content{});
```

- **Instantiation of `log_and_do`:**

```
void log_and_do(Content && param) {
  do_something(std::forward<Content>(param));
}
```

- **Instantiation of `std::forward<Content>(param)`:**

```
decltype(auto) forward(std::remove_reference_t<Content> & param) {
  return static_cast<Content &&>(param);
}
```

Example: Call with Rvalue Continued


```
decltype(auto) forward(std::remove_reference_t<Content> & param) {
  return static_cast<Content &&>(param);
}
```

- **Collapsing is not required**

```
decltype(auto) forward(Content & param) {
  return static_cast<Content &&>(param);
}
```

- **As a result `std::forward<T>(param)` yields an rvalue reference to `param` (same as `std::move`)**

```
template<typename T>
void log_and_do(T && param) {
  do_something(std::forward<T>(param));
}
```

- **Eventually `do_something(S &&)` will be called (rvalue overload)**

Inside `std::move`

- **How does `std::move` actually move objects?**
  - It doesn't!
  - It's just a simple (unconditional) cast to an rvalue reference...
  - This allows resolution of rvalue reference overloads and move-constructor/-assignment operator
- **The implementation is similar to the following:**

```
template<typename T>
decltype(auto) move(T && param) {
  return static_cast<std::remove_reference_t<T>&&>(param);
}
```

- **`std::remove_reference_t` is required to strip `param` from an lvalue reference part, otherwise the return type would still be an lvalue reference**

Summary: Type Deduction


```
template<typename T>
void f(ParamType param);
```

- **ParamType is a value/pointer type**
  - <expr> is a reference type: ignore the reference
  - Ignore const of <expr> type (outermost)
  - Pattern match <expr>’s type against ParamType to figure out T
- **ParamType is a reference**
  - <expr> is a reference type: ignore the reference
  - Pattern match <expr>’s type against ParamType to figure out T
- **ParamType is a forwarding reference (T&& / auto&&)**
  - <expr> is an lvalue: T and ParamType become lvalue references!
  - Otherwise (if <expr> is an rvalue): Rules for pointer/references apply

Deduction in Lambdas

Self-Study

- **What do you think about this code snippet?**

```
int i0 = 42;
auto missingMutable = [i0] {return i0++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() const {
    return i0++;
  }
  int i0;
};
```

- **The code won't compile as the generated operator is `const`**

- **How about now?**

```
int i1 = 42;
auto everyThingIsOk = [i1] () mutable {return i1++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() {
    return i1++;
  }
  int i1;
};
```

- **The code will compile as the generated operator is not `const`**

- **How about now?**

```
int const i2 = 42;
auto surprise = [i2] () mutable {return i2++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() {
    return i2++;
  }
  int const i2;
};
```

- **The code won't compile since `i2` is `const`**

- **How about now?**

```
int const i3 = 42;
auto srslyWhy = [i3 = i3] () mutable {return i3++;};
```

- **The compiler will generate something like this:**

```
struct CompilerKnows {
  int operator()() {
    return i3++;
  }
  int i3;
};
```

- **The init capture is deduced as if it was `auto`**

Inside `std::move`

Self-Study

Cevelop
Your C++ deserves it

OST
Ostschweizer
Fachhochschule

- **How does `std::move` actually move objects?**
  - It doesn't!
  - It's just a simple (unconditional) cast to an rvalue reference...
  - This allows resolution of rvalue reference overloads and move-constructor/-assignment operator
- **The implementation is similar to the following:**

```
template<typename T>
decltype(auto) move(T && param) {
  return static_cast<std::remove_reference_t<T>&&>(param);
}
```

- **`std::remove_reference_t` is required to strip `param` from an lvalue reference part, otherwise the return type would still be an lvalue reference**

```
template<typename T>
decltype(auto) move(T && param) {
  return static_cast<std::remove_reference_t<T> &&>(param);
}
```

- **Let's have a detailed look at a `std::move` call**
- **How should fill be implemented?**

```
struct Bottle {
  void fill(Content && liquid) {
    c = liquid;
  }
  Content c;
};
```

Copy

```
struct Bottle {
  void fill(Content && liquid) {
    c = std::move(liquid);
  }
  Content c;
};
```

Move

- **Accessing the parameter liquid is an lvalue: `std::move` is required to move the content.**

Template

```
template<typename T>
decltype(auto) move(T && param) {
  return static_cast<std::remove_reference_t<T> &&>(param);
}
```

- **Let's have a detailed look at the type deduction in the `std::move(liquid)` call**
  - `liquid`: <expr> has type `Content &&`; however, it is an lvalue!
  - T: is deduced to `Content &`
  - ParamType: becomes `Content &`

Instance

```
decltype(auto) move(Content & param) {
  return static_cast<std::remove_reference_t<Content &> &&>(param);
}
```

```
template<typename Tp>
using remove_reference_t = typename remove_reference<Tp>::type;
```

- **Template alias for specialized `remove_reference` class template**

- **Which specialization is selected? For `Content &`**

best match for
Content &

```
template<typename Tp>
struct remove_reference { typedef Tp type; };

template<typename Tp>
struct remove_reference<Tp &> { typedef Tp type; };

template<typename Tp>
struct remove_reference<Tp &&> { typedef Tp type; };
```

```
remove_reference_t<Content &> => Content
```

std::move(liquid) Resolved

```
decltype(auto) move(Content & param) {
  return static_cast<Content &&>(param);
}
```

- **What is the return type?**

```
Content && move(Content & param) {
  return static_cast<Content &&>(param);
}
```

- **A call to `std::move` is just an unconditional cast to an rvalue reference of the original type**

```
c = std::move(liquid);
```

```
struct Content {
  //...
  Content & operator=(Content && newContent);

};
```

What if `std::move` Moves an Rvalue?


```
std::move(Content{})
```

- **Type deduction**
  - `Content{}`: <expr> has type `Content`; it is an rvalue!
  - T: is deduced to `Content`
  - ParamType: becomes `Content &&`

```
decltype(auto) move(Content && param) {
  return static_cast<std::remove_reference_t<Content> &&>(param);
}
```

- **`remove_reference` strips nothing from `Content` and yields `Content`**

```
template<typename Tp>
struct remove_reference { typedef Tp type; };
```

```
decltype(auto) move(Content && param) {
  return static_cast<Content &&>(param);
}
```

Couldn't `std::move` Just Simply Cast to Rvalue Reference?

- **What if `std::move` was implemented as follows?**

```
template<typename T>
decltype(auto) move(T && param) {
  return static_cast<T &&>(param);
}
```

Incorrect

- **If it was called with an lvalue the instantiation would look as follows**

```
decltype(auto) move(Content & param) {
  return static_cast<Content & &&>(param);
}
```

- **What is `Content & &&`?**

```
decltype(auto) move(Content & param) {
  return static_cast<Content &>(param);
}
```

- **Return type of `std::move(liquid)` would be `Content &`**

Reference Collapsing

- **If references get combined in a way as seen in `std::move` so called reference collapsing happens**
- **The following happens in such cases**
  - `T & &` becomes `T &`
  - `T & &&` becomes `T &`
  - `T && &` becomes `T &`
  - `T && &&` becomes `T &&`
- **Example: This happens in the parameter of `std::move<T &>`**
  - Type of parameter `T & &&` results in `T &`