Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 5 – Perfect Forwarding

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 21.03.2019
FS2019

Recap Week 4

- **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 pointer type (e.g. `void f(T * param)`)
3. ParamType is a reference (e.g. `void f(T & param)`)
4. 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)`)

Rules for Type Deduction: Forwarding Reference Type

- **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`

Rules for decltype

- **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
}
```

- **Topics:**
  - Inside `std::move`
  - Perfect Forwarding
  - Forwarding Special Member Functions

- **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 &`**

```
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; };
```

best match for `Content &`

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

```
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 &`

Perfect Forwarding

- **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`**

Use Case: Log Wrapper

- **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.

Use Case: Log Wrapper

- **We need 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));
}
```

- **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?**
  - As you know from reference collapsing: `<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)**

Forwarding Special Member Functions

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

Duplication in Assignment

```
template<typename T, std::size_t N>
struct BoundedBuffer {
  std::array<T, N> values { };
  std::size_t start { };
  std::size_t size { };
};
```

- **Copy Assignment Operator**

```
BB & operator=(BB const & other) {
  values = other.values;
  start = other.start;
  size = other.size;
  return *this;
}
```

- **Move Assignment Operator**

```
BB & operator=(BB && other) {
  values = std::move(other.values);
  start = std::move(other.start);
  size = std::move(other.size);
  return *this;
}
```

Possible Implementation of Forward Assignment

```
BB & operator=(BB const & other) {
  values = other.values;
  start = other.start;
  size = other.size;
  return *this;
}
```

```
BB & operator=(BB && other) {
  values = std::move(other.values);
  start = std::move(other.start);
  size = std::move(other.size);
  return *this;
}
```

```
template<typename OtherType>
BoundedBuffer & operator=(OtherType && other) {
  values = std::forward<OtherType>(other).values;
  start = std::forward<OtherType>(other).start;
  size = std::forward<OtherType>(other).size;
  return *this;
}
```

- **`std::forward` makes `other` an xvalue if the argument was not an lvalue**
  - Member access of an xvalue is an xvalue too

Instances when Assigning

```
template<typename OtherType>
BoundedBuffer & operator=(OtherType && other) {
  //...
}
```

```
BoundedBuffer<int, 1> buf1{};
BoundedBuffer<int, 1> buf2{};
buf2 = buf1;
```

➡

```
BoundedBuffer & operator=(OtherType & other) {
  //...
}
```

```
BoundedBuffer<int, 1> const buf1{};
BoundedBuffer<int, 1> buf2{};
buf2 = buf1;
```

➡

```
BoundedBuffer & operator=(OtherType const & other) {
  //...
}
```

```
BoundedBuffer<int, 1> buf1{};
BoundedBuffer<int, 1> buf2{};
buf2 = std::move(buf1);
```

➡

```
BoundedBuffer & operator=(OtherType && other) {
  //...
}
```

And They Lived Happily Ever After…

- **... until the Constructors were implemented**

The good sister

```
BoundedBuffer(BoundedBuffer const & other)
  : values { other.values },
    start { other.start },
    size { other.size } {}
```

The evil sister

```
BoundedBuffer(BoundedBuffer && other)
  : values { std::move(other.values) },
    start { std::move(other.start) },
    size { std::move(other.size) } {}
```

- **After implementing the move constructor the code won't compile anymore**

```
error: use of deleted function 'BoundedBuffer<int, 1u>& BoundedBuffer<int, 1u>::operator=(const BoundedBuffer<int, 1u>&)'
```

- **The explicit move constructor deletes the implicit copy assignment**

```
BoundedBuffer & operator=(OtherType const & other) = delete;
```

Explicit Move Constructor Deletes Implicit Copy Assignment

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

Wolf in sheep's clothing

```
template<typename OtherType>
BoundedBuffer(OtherType && other)
  : values { std::forward<OtherType>(other).values },
    start { std::forward<OtherType>(other).start },
    size { std::forward<OtherType>(other).size } {}
```

- **If instead of explicitly declaring the move (and copy) constructor you create a forwarding constructor, the (implicit) copy assignment operator does not get deleted**
- **A problem arises: We now have an universal constructor (and assignment operator) that takes expressions of arbitrary type**
  - It is possible to further restrict the applicability of those operations using additional template magic
  - Details: https://akrzemi1.wordpress.com/2013/10/10/too-perfect-forwarding/
- **However, it is unlikely you will need special member functions that can be implemented in this way to be specified explicity**
  - The given implementation is what the defaults do anyway

Summary

- **There are several kind of expression types in C++ (lvalue, xvalue, prvalue)**
- **Objects/values can be copied, moved or passed by reference**
- **The compiler may omit certain copy and move operations**
- **Type deduction is not always obvious**
- **Perfect forwarding retains lvalue-/rvalue-ness of arguments**
- **Perfect forwarding can be too perfect for special member functions**
- **Good read about rvalue references and move semantics:**
  http://thbecker.net/articles/rvalue_references/section_01.html