std::forward

Defined in header <utility>

`template< class T >` `T&& forward( typename std::remove_reference<T>::type& t ) noexcept;`	(1)	(since C++11) (until C++14)
`template< class T >` `constexpr T&& forward( std::remove_reference_t<T>& t ) noexcept;`		(since C++14)
`template< class T >` `T&& forward( typename std::remove_reference<T>::type&& t ) noexcept;`	(2)	(since C++11) (until C++14)
`template< class T >` `constexpr T&& forward( std::remove_reference_t<T>&& t ) noexcept;`		(since C++14)

1) Forwards lvalues as either lvalues or as rvalues, depending on T

When `t` is a forwarding reference (a function argument that is declared as an rvalue reference to a cv-unqualified function template parameter), this overload forwards the argument to another function with the value category it had when passed to the calling function.

For example, if used in a wrapper such as the following, the template behaves as described below:

```
template<class T>
void wrapper(T&& arg)
{
    // arg is always lvalue
    foo(std::forward<T>(arg)); // Forward as lvalue or as rvalue, depending on T
}
```

- If a call to `wrapper()` passes an rvalue `std::string`, then T is deduced to `std::string` (not `std::string&`, `const std::string&`, or `std::string&&`), and `std::forward` ensures that an rvalue reference is passed to `foo`.
- If a call to `wrapper()` passes a const lvalue `std::string`, then T is deduced to `const std::string&`, and `std::forward` ensures that a const lvalue reference is passed to `foo`.
- If a call to `wrapper()` passes a non-const lvalue `std::string`, then T is deduced to `std::string&`, and `std::forward` ensures that a non-const lvalue reference is passed to `foo`.

2) Forwards rvalues as rvalues and prohibits forwarding of rvalues as lvalues

This overload makes it possible to forward a result of an expression (such as function call), which may be rvalue or lvalue, as the original value category of a forwarding reference argument.

For example, if a wrapper does not just forward its argument, but calls a member function on the argument, and forwards its result:

```
// transforming wrapper
template<class T>
void wrapper(T&& arg)
{
    foo(forward<decltype(forward<T>(arg).get())>(forward<T>(arg).get()));
}
```

where the type of arg may be

```
struct Arg
{
    int i = 1;
    int  get() && { return i; } // call to this overload is rvalue
    int& get() &  { return i; } // call to this overload is lvalue
};
```

Attempting to forward an rvalue as an lvalue, such as by instantiating the form (2) with lvalue reference type T, is a compile-time error.

Notes

See template argument deduction for the special rules behind forwarding references (T&& used as a function parameter) and forwarding references for other detail.

Parameters

t - the object to be forwarded

Return value

`static_cast<T&&>(t)`

Example

This example demonstrates perfect forwarding of the parameter(s) to the argument of the constructor of class T. Also, perfect forwarding of parameter packs is demonstrated.

Run this code

```
#include <iostream>
#include <memory>
#include <utility>

struct A {
    A(int&& n) { std::cout << "rvalue overload, n=" << n << "\n"; }
    A(int& n)  { std::cout << "lvalue overload, n=" << n << "\n"; }
};

class B {
public:
    template<class T1, class T2, class T3>
    B(T1&& t1, T2&& t2, T3&& t3) :
        a1_{std::forward<T1>(t1)},
        a2_{std::forward<T2>(t2)},
        a3_{std::forward<T3>(t3)}
    {
    }

private:
    A a1_, a2_, a3_;
};

template<class T, class U>
std::unique_ptr<T> make_unique1(U&& u)
{
    return std::unique_ptr<T>(new T(std::forward<U>(u)));
}

template<class T, class... U>
std::unique_ptr<T> make_unique2(U&&... u)
{
    return std::unique_ptr<T>(new T(std::forward<U>(u)...));
}

int main()
{
    auto p1 = make_unique1<A>(2); // rvalue
    int i = 1;
    auto p2 = make_unique1<A>(i); // lvalue

    std::cout << "B\n";
    auto t = make_unique2<B>(2, i, 3);
}
```

Output:

```
rvalue overload, n=2
lvalue overload, n=1
B
```

```
rvalue overload, n=2
lvalue overload, n=1
rvalue overload, n=3
```

Complexity

Constant

See also

move (C++11)	obtains an rvalue reference (function template)
move_if_noexcept (C++11)	obtains an rvalue reference if the move constructor does not throw (function template)