

ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



DECLTYPE

```
const int i = 0; // decltype(i) - const int
```

```
bool f (const Widget &w);    // decltype(w) - const Widget&  
                             // decltype(f) - bool (const Widget&)
```

```
struct Point{  
    int x, y;    // decltype(Point::x) - int  
};              // decltype(Point::y) - int
```

```
template <typename T>  
class vector{  
public:  
    ...  
    T& operator[](size_t index);  
}
```

```
vector<int> v;          // decltype(v) - vector<int>  
if(v[0] == 0) ...      // decltype(v[0]) - int&
```

DECLTYPE

// C++11 - trailing return type

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

    return c[i]; //Return type - ???
}
```

// C++14 - deducing type

```
template <typename Container, typename Index>
auto authAndAccess(Container &c, Index i) {
    authenticateUser();
    return c[i]; //Return type - ???
}
```


DECLTYPE

// C++11 - trailing return type

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

    return c[i]; //Return type - T&
}
```

// C++14 - deducing type

```
template <typename Container, typename Index>
auto authAndAccess(Container &c, Index i) {
    authenticateUser();
    return c[i]; //Return type - T
}
```

DECLTYPE

// C++14 - deducing type

```
template <typename Container, typename Index>
auto authAndAccess(Container &c, Index i) {
    authenticateUser();
    return c[i]; //Return type - T
}
```

```
std::deque<int> d;
```

```
...
```

```
authAndAccess(d, 5) = 10; //Compilation error!
```

// C++14 - deducing type by decltype

```
template <typename Container, typename Index>
decltype(auto) authAndAccess(Container &c, Index i) {
    authenticateUser();
    return c[i]; //Return type - T&
}
```

VARIABLE DECLARATION

```
int a = 0;
```

```
const int& cref = a;
```

```
auto val1 = cref; //auto - ???
```

```
decltype(auto) val2 = cref; //decltype - ???
```


VARIABLE DECLARATION

```
int a = 0;
```

```
const int& cref = a;
```

```
auto val1 = cref; //auto - int
```

```
decltype(auto) val2 = cref; //decltype - const int&
```

DECLTYPE

```
decltype(auto) f1(){  
    int x = 0;  
    ...  
    return x;    //decltype(x) -> int  
}  
  
decltype(auto) f2(){  
    int x = 0;  
    ...  
    return (x); //decltype((x)) -> int&  
}
```

- For lvalue expressions of type T other than names, **decltype** always reports a type of $T\&$.

STD::MOVE

```
//C++11
template<typename T>
typename remove_reference<T>::type&&
move(T&& param)
{
    using ReturnType =
        typename remove_reference<T>::type&&;
    return static_cast<ReturnType>(param);
}
```

```
//C++14
template<typename T>
decltype(auto) move(T&& param)
{
    using ReturnType = remove_reference_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

STD::FORWARD

```
void process(const Widget& lvalArg);  
void process(Widget&& rvalArg);  
  
template<typename T>  
void logAndProcess(T&& param)  
{  
    auto now = std::chrono::system_clock::now();  
    makeLogEntry("Call process", now);  
    process(std::forward<T>(param));  
}
```

«Conditional cast»



```
Widget w;  
...  
logAndProcess(w);           //Call with lvalue  
logAndProcess(std::move(w)); //Call with rvalue
```

STD::MOVE & STD::FORWARD

- **std::move** performs an *unconditional* cast to an rvalue. In and of itself, it doesn't move anything.
- **std::forward** casts its argument to an rvalue only if that argument is bound to an rvalue.
- Neither **std::move**, nor **std::forward** do anything at runtime.

DISTINGUISH UNIVERSAL REFERENCES FROM RVALUE-REFERENCES.

```
void f(Widget &&param); // ???
```

```
Widget&& var1 = Widget(); // ???
```

```
auto&& var2 = var1; // ???
```

```
template <typename T>  
void f(std::vector<T>&& param); // ???
```

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

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

DISTINGUISH UNIVERSAL REFERENCES FROM RVALUE-REFERENCES.

```
void f(Widget &&param); // rvalue-reference
```

```
Widget&& var1 = Widget(); // rvalue-reference
```

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

```
template <typename T>  
void f(std::vector<T>&& param); // rvalue-reference
```

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

```
template <typename T>  
void f(const T&& param); // rvalue-reference
```

UNIVERSAL REFERENCE

```
template <class T,  
          class Allocator = allocator<T>>  
class vector {  
public:  
    // ...  
    void push_back(T&& x); // rvalue-reference  
    // ...  
};
```

- Type deduction.
- The form of the type declaration: "T&&".

THINGS TO REMEMBER

- If a function template parameter has type `T&&` for a deduced type `T`, or if an object is declared using `auto&&`, the parameter or object is a universal reference.
- If the form of the type declaration isn't precisely `type&&`, or if type deduction does not occur, `type&&` denotes an rvalue reference.
- Universal references correspond to rvalue references if they're initialized with rvalues.

```
Widget makeWidget(){  
    Widget w;  
    ...  
    return w;  
}
```

```
Widget makeWidget(){  
    Widget w;  
    ...  
    return std::move(w);  
}
```

???

```
Widget makeWidget(){  
    Widget w;  
    ...  
    return w;  
}
```

```
Widget makeWidget(){  
    Widget w;  
    ...  
    return std::move(w);  
}
```

Bad code!
NRVO doesn't work.



Never apply `std::move` or `std::forward` to local objects if they would otherwise be eligible for the return value optimization.

REFERENCE COLLAPSING

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

```
Widget widgetFactory(); // Function returns rvalue-object
```

```
Widget w; // lvalue  
f(w);     // Call with lvalue;  
          // type of T - ???
```

```
f(widgetFactory()); // Call with rvalue;  
                    // type of T - ???
```

REFERENCE COLLAPSING

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

```
Widget widgetFactory(); // Function returns rvalue-object
```

```
Widget w; // lvalue  
f(w);     // Call with lvalue;  
          // type of T - Widget&
```

```
f(widgetFactory()); // Call with rvalue;  
                    // type of T - Widget
```

REFERENCE COLLAPSING

```
int x;
```

```
...
```

```
auto& & rx = x; //Error! can't declare reference to reference
```

```
template <typename T>
```

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

```
Widget w;
```

```
f(w); // T -> Widget& →
```

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

↓ **How???**

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


REFERENCE COLLAPSING RULES

T& &	→	T&
T& &&	→	T&
T&& &	→	T&
T&& &&	→	T&&

STD::FORWARD

```
template <typename T>
void f(T&& param){
    ...
    someFunc(std::forward<T>(param));
}
```


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

```
Widget w; // lvalue
f(w);     // Call with lvalue;
          // type of T - Widget&
```

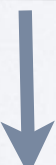
```
f(widgetFactory()); // Call with rvalue;
                    // type of T - Widget
```

STD::FORWARD CALL WITH LVALUE

```
template <typename T>
Widget& && forward(remove_reference_t<Widget&>& param){
    return static_cast<Widget& &&>(param);
}
```



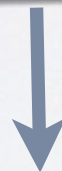
```
template <typename T>
Widget& && forward(Widget& param){
    return static_cast<Widget& &&>(param);
}
```



```
template <typename T>
Widget& forward(Widget& param){
    return static_cast<Widget&>(param);
}
```


STD::FORWARD CALL WITH RVALUE

```
template <typename T>
Widget&& forward(remove_reference_t<Widget>& param){
    return static_cast<Widget&&>(param);
}
```



```
template <typename T>
Widget&& forward(Widget& param){
    return static_cast<Widget&&>(param);
}
```

I.TEMPLATE INSTANTIATION

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

```
Widget widgetFactory(); // Function returns rvalue
```

```
Widget w; // lvalue  
f(w);     // Call with lvalue;  
          // type of T - Widget&
```

```
f(widgetFactory()); // Call with rvalue;  
                    // type of T - Widget
```

2. AUTO TYPE GENERATION

Widget widgetFactory(); // Function returns rvalue

Widget w; // lvalue

`auto&& w1 = w;` → `Widget& && w1 = w;` → `Widget& w1 = w;`

`auto&& w2 = widgetFactory();` → `Widget&& w2 = widgetFactory();`

3. CREATION AND USE OF **TYPEDEFS** AND ALIAS DECLARATIONS

```
template <typename T>  
class Widget{  
public:  
    typedef T&& RvalueRefToT;  
};
```

Widget<int&> w;

→ typedef int& && RvalueRefToT;

↓
typedef int& RvalueRefToT;

4. DECLTYPE

```
auto func(int& param) -> decltype(param)&;
```

PERFECT FORWARDING

```
struct X{  
    X(const int&, int&){}  
};
```

```
struct W{  
    W(int&, int&){}  
};
```

```
struct Y{  
    Y(int&, const int&){}  
};
```

```
struct Z{  
    Z(const int&, const int&){}  
};
```

```
template <typename T, typename A1, typename A2>  
T* factory(A1& a1, A2& a2){  
    return new T(a1, a2);  
}
```

```
int a = 4, b = 5;  
W* pw = factory<W>(a,b); // Ok.  
X* pw = factory<X>(2,b); // Error.  
Y* pw = factory<Y>(a,2); // Error.  
Z* pw = factory<Z>(2,2); // Error.
```


PERFECT FORWARDING

```
struct X{  
    X(const int&, int&){}  
};
```

```
struct W{  
    W(int&, int&){}  
};
```

```
struct Y{  
    Y(int&, const int&){}  
};
```

```
struct Z{  
    Z(const int&, const int&){}  
};
```

```
template <typename T, typename A1, typename A2>  
T* factory(A1&& a1, A2&& a2){  
    return new T(std::forward<A1>(a1), std::forward<A2>(a2));  
}
```

```
int a = 4, b = 5;  
W* pw = factory<W>(a,b); // Ok.  
X* pw = factory<X>(2,b); // Ok.  
Y* pw = factory<Y>(a,2); // Ok.  
Z* pw = factory<Z>(2,2); // Ok.
```

PERFECT FORWARDING

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

```
// Variadic template
template <typename... Ts>
void fwd(Ts&&... params){
    f(std::forward<Ts>(params)...);
}
```