

AUTO & DECLTYPE

auto and decltype specifiers by Prasanna Ghali

Plan for Today

2

- How does `auto` work
- How does `decltype` work

auto: The Basics (1 / 7)

3

- We write code like this all the time ...

```
std::vector<int> vi;  
// ...  
  
for (std::vector<int>::const_iterator cit = std::cbegin(vi);  
     cit != std::cend(vi);  
     ++cit) {  
    std::cout << *cit;  
}
```

auto: The Basics (2/7)

4

- Next, consider this code ...

```
std::vector<int> vi;  
// ...  
for (std::vector<int>::iterator it = std::begin(vi);  
     it != std::end(vi);  
     ++it) {  
    std::cin >> *it;  
}
```

- Quite annoying for programmer to explicitly write type of variable `it` when compiler knows type `it` should be!!!

auto: The Basics (3/7)

5

- Since C++11, language shares with programmer type of initializing expression `std::begin(vi)`
- Since C++11, you can declare variable without specifying its type by using `auto`

```
std::vector<int> vi;  
// ...  
for (auto it = std::begin(vi); it != std::end(vi); ++it) {  
    std::cin >> *it;  
}
```

auto: The Basics (4/7)

6

```
double f(); // f() returns value of type double
auto d {f()}; // f() and d have type double
auto i {42}; // 42 and i have type int
auto flag{false}; // false and flag has type int

// additional qualifiers are allowed
static auto pi = 3.1428;

std::vector<std::string> vs;
// using auto is especially useful where the type
// is a pretty long and/or complicated expression
// pos has type std::vector<std::string>::iterator
auto pos = std::begin(vs);
```

auto: The Basics (5/7)

7

- **auto** doesn't change fact that C++ is strongly typed

```
double f();  
auto d {f()}; // f() and d have type double
```

- ▣ Variable **d** has result type of expression in initializer, and this type will never change afterward
- ▣ In Python, variable **d** will have *dynamic type*: an assignment to **d** can change type of **d** to that of assigned expression

auto: The Basics (6/7)

8

- Initializer is required since type of variable declared with **auto** is deduced from its initializer

```
auto a;           // error: don't know type of a
a = f(x, y);      // too late ...

// we can declare multiple auto variables in same statement
// as long as all initializers are expressions of same type
auto i {2*7.5}, j {std::sqrt(3.9)}; // ok: both are double
auto m {2*4},    n {std::sqrt(3.9)}; // error: m is int,
                                     // n is double

auto o {2*4}, p; // error: p not initialized
auto q {f(i, j)}; // type of result of f(i, j)
```


auto: The Basics (7/7)

9

- Can qualify **auto** with **const** and/or reference qualifiers:

```
auto i{2*7}; // i is int variable initialized to 14
```

```
const auto ci {i}; // ci's type deduced as int const
```

```
auto const ci2 {i}; // ci2's type deduced as int const
```

```
auto& ri {i}; // ri's type deduced as int&
```

```
auto const& rci {i}; // rci's type deduced as int const&
```

```
const auto& rci2 {i}; // same as rci
```

auto: Beyond The Basics (1 / 4)

10

□ Consider this example of using **auto**:

```
int x{};           // x is int initialized to 0
int const& rcx{x};  // rcx is read-only reference of x
x = 21;
auto something{rcx}; // what is the type of something?
++x;
std::cout << something; // what is value written to stdout?
++something;           // will this compile?
```

Since declared type of **rcx** is **int const&** and initializing expression for **something** is **rcx**, you might think that type of **something** is **int const&**.

Not so.

It turns out type of **something** is deduced as **int**.

auto: Beyond The Basics (2/4)

11

□ Consider more scenarios that use **auto**:

```
char str[6] {"hello"}; // str is array of 6 char elements

auto str2 {"hello"}; // what is type of str2?
auto str3 = "world"; // what is type of str3?
auto str4 {str};      // what is type of str4?
auto ia {1,2,3,4};    // what is type of ia?
```

Both "hello" and "world" are of type `char const [6]` which evaluates to `char const*`; therefore `str2` and `str3` are both deduced to have type `char const*`.
`str` evaluates to type `char*`; therefore `str4` has type `char*`.
With braced initializers `{1,2,3,4}`, `ia`'s type is deduced as `std::initializer_list<int>`.

auto: Beyond The Basics (3/4)

12

□ What about expressions with `auto&&`?

```
double f(int x, int y) { return static_cast<double>(x+y); }  
template <typename T> T& g(T& x) { return ++x; }
```

*// which of following declaration statements compile
// and if they compile, what is type of variable?*

```
auto    i {42};           // i has type int  
auto&& rri {f(i, i)};      // what is type of rri?  
auto&& rri2 {42};          // what is type of rri2?  
auto&& rri3 {i};           // what is type of rri3?  
auto&& rri4 {g(42)};       // what is type of rri4?  
auto&& rri5 {g(i)};        // what is type of rri5?
```

auto: Beyond The Basics (4/4)

13

□ What about expressions with `auto&&`?

```
double f(int x, int y) { return static_cast<double>(x+y); }  
template <typename T> T& g(T& x) { return ++x; }
```

*// which of following declaration statements compile
// and if they compile, what is type of variable?*

```
auto    i {42};           // i has type int  
auto&& rri {f(i, i)};      // rri deduced as rvalue reference  
auto&& rri2 {42};          // rri2 deduced as rvalue reference  
auto&& rri3 {i};           // rri3 deduced as lvalue reference  
auto&& rri4 {g(42)};       // error: 42 is not an lvalue  
auto&& rri5 {g(i)};        // rri5 deduced as lvalue reference
```

How to Understand `auto` Declarations? (1 / 6)

14

- In previous examples, we saw that `auto` type specifier is used in number of places to deduce type of variable from its initializer
- `auto` type deduction is template type deduction
- There is direct algorithmic transformation from template type deduction to `auto` type deduction

How to Understand **auto** Declarations? (2/6)

15

```
auto          x {42}; // what is type deduced for x???  
const auto    cx {x}; // what is type deduced for cx???  
const auto&    rx {x}; // what is type deduced for rx???
```

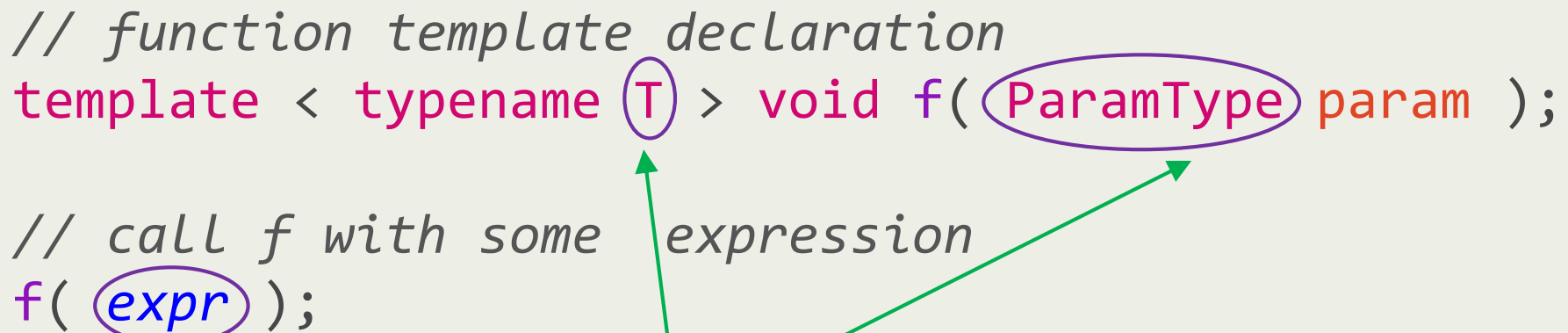
- To deduce types for **x**, **cx**, and **rx** in these examples, compilers act as if there were a function template for each declaration as well as a call to that template with corresponding initializing expression

How to Understand **auto** Declarations? (3/6)

16

- Type deduced for **identifier** is type deduced for **param** through template type deduction

```
// function template declaration  
template < typename T > void f( ParamType param );  
  
// call f with some expression  
f( expr );
```



*type specifier with **auto** identifier = **initializer**;*

How to Understand **auto** Declarations? (4/6)

17

```
auto x {42}; // what is type deduced for x???
```



```
template <typename T> // conceptual template for  
void func_for_x(T param); // deducing x's type  
func_for_x(42); // conceptual call: param's  
// deduced type is x's type
```


How to Understand auto Declarations? (6/6)

19

```
auto      x {42}; // what is type deduced for x
template <typename T> // conceptual template for
void func_for_x(T param); // deducing x's type
func_for_x(42); // conceptual call: param's
                // deduced type is x's type

const auto cx {x}; // what is type deduced for cx
template <typename T> // conceptual template for
void func_for_cx(const T param); // deducing cx's type
func_for_cx(x); // conceptual call: param's
                // deduced type is cx's type

const auto& rx {x}; // what is type deduced for rx
template <typename T> // conceptual template for
void func_for_rx(const T& param); // deducing rx's type
func_for_rx(x); // conceptual call: param's
                // deduced type is rx's type
```

auto Type Deduction: Cases

20

- Just as with template type deduction, **auto** type deduction has three cases to consider:
 - ▣ *type specifier* is pointer or reference type
 - ▣ *type specifier* is neither a pointer nor reference
 - ▣ *type specifier* is forwarding reference

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

// call f with some expression
f( expr );
```

type specifier with auto identifier = initializer;

auto Type Deduction: Examples

(1 / 2)

21

- Three cases to consider:
 - *type specifier* is pointer or reference type
 - *type specifier* is neither a pointer nor reference
 - *type specifier* is forwarding reference

```
// case 2
auto x {27};           // x: ???
auto const cx {x};     // cx: ???

// case 1
auto const& rx {x};    // rx: ???

// case 3
auto&& ref1 {x};       // ref1: ???
auto&& ref2 {cx};      // ref2: ???
auto&& ref3 {27};      // ref3: ???
```

auto Type Deduction: Examples

(2/2)

22

```
// arrays
char const name[] {"hello"};
auto arr1 {name};           // arr1: ???
auto& arr2 {name};          // arr2: ???

// functions
void func(int, double);
auto fun1 {func};           // fun1: ???
auto& fun2 {func};          // fun2: ???
```

auto Type Deduction: Exception

(1 / 5)

23

- **auto** works like template type deduction
- Except for one way they differ
- Start with observation that if you want to initialize an **int**
 - ▣ C++98 gives 2 choices
 - ▣ C++11 gives 4 choices

```
// C++98 initialization syntax  
int x1 = 27;  
int x2 (27);
```

```
// C++11 initialization syntax  
int x1 = 27;  
int x2 (27);  
int x2 = {27};  
int x2 {27};
```

auto Type Deduction: Exception

(2/5)

24

- We can initialize `ints` in 4 ways but replacing `int` with `auto` is not *always* equivalent!!!

```
// C++11 initialization syntax
```

```
int x1 = 27;  
int x2 (27);  
int x3 = {27};  
int x4 {27};
```

```
// All declarations will compile after  
// substitution of int with auto
```

```
auto x1 = 27;  
auto x2 (27);  
auto x3 = {27};  
auto x4 {27};
```


auto Type Deduction: Exception

(3/5)

25

- When initializer for **auto**-declared variable is enclosed in braces, deduced type is `std::initializer_list`

```
auto x1 = 27;    // type is int, value is 27
auto x2 (27);    // type is int, value is 27
auto x3 = {27};  // type is std::initializer_list<int>
                  // value is {27}
auto x4 {27};    // type is std::initializer_list<int>
                  // value is {27}
```

auto Type Deduction: Exception

(4/5)

26

- Treatment of braced initializers is only way in which **auto** type deduction and template type deduction differ!!!

```
// we know x's type is deduced as std::initializer_list<int>  
auto x = {11, 23, 9};
```

```
// template equivalent to x's declaration  
template <typename T> void f(T param);  
f({11, 23, 9}); // error: cannot deduce type for T
```

```
// template for which template type deduction will work  
template <typename T> void g(std::initializer_list<T> param);  
g({11, 23, 9}); // ok: T deduced as int and  
               // param's type is initializer_list<int>
```

auto Type Deduction: Exception

(5/5)

27

- So only real difference between `auto` and template type deduction is that `auto` assumes braced initializer represents `std::initializer_list` but template type deduction doesn't!!!
- Remember that if you declare a variable using `auto` and you initialize it with braced initializer, deduced type will always be `std::initializer_list`

Prefer **auto** Over Explicit Type Declarations (1 / 9)

28

□ Less verbose declarations

```
// print values of all elements in range [b, e)
template <typename It>
void print(It b, It e) {
    while (b != e) {
        typename std::iterator_traits<It>::value_type val = *b;
        std::cout << val << ' ';
        ++b;
    }
    std::cout << "\n";
}

std::vector<std::string> vs{"a", "b", "c", "d"};
print(std::begin(vs), std::end(vs));

std::deque<int> di{1, 2, 3, 4, 5, 6};
print(std::begin(di), std::end(di));
```

Prefer **auto** Over Explicit Type Declarations (2/9)

29

□ Less verbose declarations

```
// print values of all elements in range [b, e)
template <typename It>
void print(It b, It e) {
    while (b != e) {
        auto val = *b;
        std::cout << val << ' ';
        ++b;
    }
    std::cout << "\n";
}

std::vector<std::string> vs{"a", "b", "c", "d"};
print(std::begin(vs), std::end(vs));

std::deque<int> di{1, 2, 3, 4, 5, 6};
print(std::begin(di), std::end(di));
```

Prefer **auto** Over Explicit Type Declarations (3/9)

30

□ Prevents uninitialized variables

```
int i1;      // potentially uninitialized
auto i2;     // error: initializer required
auto i3 {0}; // ok: i3's value is well defined
```

```
// use of auto prevents uninitialized local variables
template <typename It>
void print(It b, It e) {
    while (b != e) {
        auto val = *b;
        std::cout << val << ' ';
        ++b;
    }
    std::cout << "\n";
}
```

Prefer **auto** Over Explicit Type Declarations (4/9)

31

□ Prevents “type shortcuts”

```
std::vector<std::string> vs{"a", "b", "c", "d"};

// fairly common type shortcut that might cause
// problems on 64-bit machines ...
unsigned sz = vs.size();

// programmer should have used explicit type ...
std::vector<std::string>::size_type sz2 = vs.size();

// however, many programmers are unaware of the official
// return type of vs.size() ...
// using auto ensures you don't have to spend time
// remembering official return type and understanding
// 32-bit vs 64-bit issues ...
auto sz3 = vs.size(); // sz3's type is official return type ...
```

Prefer **auto** Over Explicit Type Declarations (5/9)

32

□ Prevents “type mismatches”

```
std::map<std::string,int> m { {"tagged",6}, {"a",1}, {"this",4} };

// insert lots more key/value pairs into m

// let's print all key/value pairs ...
for (std::pair<std::string,int> const& p : m) {
    std::cout << p.first << ' ' << p.second << '\n';
}
// there's a problem in the loop - do you see it?

// unintentional type mismatches can be autoed away!!!
for (auto const& p : m) {
    std::cout << p.first << ' ' << p.second << '\n';
}
// not only is the loop more efficient, it's also easier to type!!!
```


Prefer **auto** Over Explicit Type Declarations (6/9)

33

- Necessary to represent types known only to compilers

```
// lam is lambda that returns true when x is even integer  
auto lam = [](int x) -> bool { return x%2 == 0; };  
  
bool flag = lam(33); // flag is false
```

Prefer **auto** Over Explicit Type Declarations (7/9)

34

- Necessary for deducing return type ...

```
// return type depends on template type parameters ...  
template <typename T1, typename T2>  
??? sum(T1 t1, T2 t2) {  
    return t1 + t2;  
}
```

```
auto s1 = sum(1, 1.1); // type of s1 must be double  
auto s2 = sum(1.1, 1); // type of s2 must be double  
auto s3 = sum(1.f, 1); // type of s3 must be float  
auto s4 = sum(1UL, 2U); // type of s4 must be unsigned long
```

Prefer **auto** Over Explicit Type Declarations (8/9)

35

- C++14 permits use of **auto** to indicate function's return type should be deduced using template type deduction

```
// since C++14, return type deduced by  
// template type deduction ...
```

```
template <typename T1, typename T2>  
auto sum(T1 t1, T2 t2) {  
    return t1 + t2;  
}
```

```
auto s1 = sum(1, 1.1); // type of s1 is double  
auto s2 = sum(1.1, 1); // type of s2 is double  
auto s3 = sum(1.f, 1); // type of s3 is float  
auto s4 = sum(1UL, 2U); // type of s4 is unsigned long
```

Prefer **auto** Over Explicit Type Declarations (9/9)

36

- **auto** prevents uninitialized variables
- **auto** declarations are less verbose
- **auto** prevents “type shortcuts”
- **auto** prevents “type mismatches”
- **auto** necessary to represent types known only to compilers [closures or lambdas]
- **auto** necessary to deduce return type using template type deduction

auto Type Specifier: Summary

37

- **auto** type deduction is usually same as template type deduction, but **auto** type deduction assumes braced initializer represents `std::initializer_list` while template deduction doesn't
- **auto** in a function return type implies template type deduction, not **auto** type deduction

decltype Specifier

38

- **auto** avoids need to write out type of variable, but it doesn't allow use of type of that variable
- **decltype** allows us to express precise type of an expression or declaration

```
auto          i{0};  
  
decltype(i)    j{i}; // j has type ???  
decltype((i)) k{i}; // k has type ???
```

decltype Specifier: The Basics

(1 / 5)

39

- If `e` is name of entity [variable, function, enumerator, class member] `decltype(e)` yields *declared type* of that entity
- Otherwise, if `e` is expression, `decltype(e)` produces type that reflects expression's *type and value category*
 - ▣ If `e` is *lvalue* of type `T`, `decltype(e)` produces `T&`
 - ▣ If `e` is *prvalue* of type `T`, `decltype(e)` produces `T`
 - ▣ If `e` is *xvalue* of type `T`, `decltype(e)` produces `T&&`

decltype Specifier: The Basics

(2/5)

40

- Given name or expression, **decltype** tells you type of name or expression [without evaluating expression]

```
auto i{0}, &ri{i}, *pi{&i};  
auto const ci{0}, &cj{ci};
```

// what decltype tells you is exactly what you'd predict

```
decltype(i)      j{0};           // j has type ???  
decltype((i))    k{i};           // k has type ???  
decltype(ri)     m{j};           // m has type ???  
decltype(ci)     x{0};           // x has type ???  
decltype(cj)     y{j};           // y has type ???  
decltype(ci+10)  z{11};          // z has type ???  
decltype(std::move(i)) n{12};    // n has type ???  
decltype(cj)     xx{10};         // xx has type ???  
decltype(ri+10)  yy;             // yy has type ???  
decltype(*pi)    zz{i};          // zz has type ???
```


decltype Specifier: The Basics

(3/5)

41

```
// decltype parrots back exact type of name or expr you give it

int const i{};           // decltype(i) is int const

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

Point p;                // decltype(p) is Point
bool f(Point const& s); // decltype(s) is Point const&
                        // decltype(f) is bool(Point const&)
if (f(p)) ...           // decltype(f(p)) is bool

std::vector<int> v;      // decltype(v) is std::vector<int>

if (v[0] == 0) ...      // decltype(v[0]) is int&
```

decltype Specifier: The Basics

(4/5)

42

- **decltype** looks like a function and returns type of an expression

```
// ok: use template type deduction  
template <typename T>  
auto incr(T x) {  
    return x+1;  
}
```

```
// ok: x has type int  
decltype(incr(10)) x {11};
```

```
// ok: y has type unsigned int  
decltype(incr(10U)) y = 10;
```

decltype Specifier: The Basics

(5/5)

43

```
void g(std::string&& s) {  
    // check the type of s:  
    std::is_lvalue_reference<decltype(s)>::value;    // true or false  
    std::is_rvalue_reference<decltype(s)>::value;    // true or false  
    std::is_same<decltype(s), std::string&>::value;  // true or false  
    std::is_same<decltype(s), std::string&&>::value; // true or false  
  
    // check value category of s used as expression:  
    std::is_lvalue_reference<decltype((s))>::value; // true or false  
    std::is_rvalue_reference<decltype((s))>::value; // true or false  
    std::is_same<decltype((s)), std::string&>::value; // true or false  
    std::is_same<decltype((s)), std::string&&>::value; // true or false  
}
```

decltype Specifier: Deducing Function Return Types (1 / 3)

44

- Primary use of `decltype` in C++11 is declaring function templates where function's return type depends on its parameter types

```
// will this compile?  
template <typename T1, typename T2>  
decltype(x+y) Add(T1 x, T2 y) {  
    return x+y;  
}
```

decltype Specifier: Deducing Function Return Types (2/3)

45

```
// before C++14, it is only possible to let compiler  
// determine return type by making function's  
// implementation part of its declaration ..
```

```
template <typename T1, typename T2>  
auto Add(T1 x, T2 y) -> decltype(x+y) {  
    return x+y;  
}
```

```
int x = 10; double y = 20.2;  
auto z = Add(x, y); // type of z ???
```

```
std::string sx = "hello"; char sy[]="world";  
auto sz = Add(sx, sy); // type of sz ???
```

decltype Specifier: Deducing Function Return Types (3/3)

46

- **decltype** is important in places where explicit type is required

```
// how to get this code fragment to compile?  
std::vector<int>      v1{1, 2, 3, 4};  
std::vector<double>  v2{1.1, 2.2, 3.3, 4.4, 5.5};  
std::vector<??>      v3 = v1+v2;  
// now v3 is {2.1, 4.2, 6.3, 8.4}
```

decltype Specifier: Deducing Function Return Types (4/5)

47

- Suppose we'd like to write function that takes container that supports indexing and returns result of indexing operation:

```
// std::vector<T>::op[] returns T&  
// std::deque<T>::op[] returns T&  
// unlike other containers,  
// std::vector<bool>::op[] doesn't return bool&  
template <typename Cont, typename Index>  
auto access(Cont& c, Index i) {  
    return c[i];  
}  
// incorrect since with auto return type,  
// compilers employ template type deduction!!!
```

decltype Specifier: Deducing Function Return Types (5/5)

48

- **decltype** is required for correct definition:

```
// std::vector<T>::op[] returns T&
// std::deque<T>::op[] returns T&
// unlike other containers,
// std::vector<bool>::op[] doesn't return bool&
template <typename Cont, typename Index>
auto access(Cont& c, Index i) -> decltype(c[i]) {
    return c[i];
}
```


decltype(auto) Specifier (1 / 2)

49

- Since C++14, we can declare `auto` variables that have same type as with `decltype`
 - ▣ `auto` specifies that type is to be deduced
 - ▣ `decltype` says that `decltype` rules should be used during the deduction

decltype(auto) Specifier (2/2)

50

```
int i {21};  
int const &ri {i};  
  
// auto type deduction  
auto x = ri;           // x has type ???  
  
// decltype type deduction  
decltype(auto) y = ri; // y has type ???
```

```
std::vector<int> v {21};  
auto xx = v[0];           // what is type of xx???  
decltype(auto) yy = v[0]; // what is type of yy???
```

decltype(auto) Specifier:

Deducing Function Return Types (1)

51

```
// std::vector<T>::op[] returns T&  
// std::deque<T>::op[] returns T&  
// unlike other containers,  
// std::vector<bool>::op[] doesn't return bool&  
template <typename Cont, typename Index>  
auto access(Cont& c, Index i) -> decltype(c[i]) {  
    return c[i];  
}
```

```
// auto specifies that type is to be deduced  
// decltype says decltype rules to be used  
// during the deduction  
template <typename Cont, typename Index>  
decltype(auto) access(Cont& c, Index i) {  
    return c[i];  
}
```

decltype(auto) Specifier: Deducing Function Return Types (2)

52

- Want this to work for both *lvalues* and *rvalues*

```
// auto specifies that type is to be deduced
// decltype says decltype rules to be used
// during the deduction
template <typename Cont, typename Index>
decltype(auto) access(Cont& c, Index i) {
    return c[i];
}
```

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

decltype Specifier: Edge Case

53

- Something to worry about since C++14:

```
decltype(auto) f1() {  
    int x = 0;  
    return x;  
}  
  
decltype(auto) f2() {  
    int x = 0;  
    return ((x));  
}
```

decltype Specifier: Summary

54

- `decltype` almost always yields type of variable or expression without any modifications
- For lvalue expressions of type `T` other than names, `decltype` always reports type of `T&`
- Since C++14, `decltype(auto)` deduces type from its initializer, but it performs type deduction using `decltype` rules