AUTO TYPE DEDUCTION & UNDERSTANDING DECLTYPE

auto and decltype

by Prasanna Ghali

Plan for Today

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

Template Type Deduction (1/3)

- Entire discussion is based on the excellent material presented here

Template Type Deduction (2/3)

- Consider function template and call to that function template:

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

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

- *Template type deduction* is process during compilation when compilers use `expr` to deduce types for `T` and `ParamType`

Template Type Deduction (3/3)

- *Template type deduction* is process during compilation when compilers use `expr` to deduce types for `T` and `ParamType`
- Three cases to consider:
  - `ParamType` is pointer or reference type
  - `ParamType` is neither pointer nor reference
  - `ParamType` is forwarding reference

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

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

ParamType: Pointer/Reference (1/8)

- If expr's type is reference, ignore reference part and then pattern-match expr's type against ParamType to determine T

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

f(expr);
```

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

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: ???, param: ???
f(cx); // T: ???, param: ???
f(rx); // T: ???, param: ???
```

ParamType: Pointer/Reference (2/8)

- If expr's type is reference, ignore reference part and then pattern-match expr's type against ParamType to determine T

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

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);   // T: int, param: int&
f(cx);  // T: const int, param: const int&
f(rx);  // T: const int, param: const int&
```

ParamType: Pointer/Reference (3/8)

- ParamType's type now changes from T& to T const&
- If expr's type is reference, ignore reference and pattern-match expr's type against ParamType to determine T

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

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: ???, param: ???
f(cx); // T: ???, param: ???
f(rx); // T: ???, param: ???
```

ParamType: Pointer/Reference (4/8)

- ParamType's type now changes from T& to T const&
- If expr's type is reference, ignore reference and pattern-match expr's type against ParamType to determine T

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

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to
                    // x as a const int

// what are deduced types for param and T?
f(x);  // T: int, param: int const&
f(cx); // T: int, param: int const&
f(rx); // T: int, param: int const&
```

ParamType: Pointer/Reference (5/8)

- ParamType's type is T*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

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

int x{27};            // x is an int
int const *px{&x}; // px is pointer to x
                      // as a const int

// what are deduced types for param and T?
f(&x);  // T: ???, param: ???
f(px); // T: ???, param: ???
```

ParamType: Pointer/Reference (6/8)

- ParamType's type is T*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

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

int x{27};             // x is an int
int const *px{&x};  // px is pointer to x
                       // as a const int

// what are deduced types for param and T?
f(&x);  // T: int, param: int*
f(px); // T: int const, param: int const*
```

ParamType: Pointer/Reference (7/8)

- ParamType's type is T const*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

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

int x{27};           // x is an int
int const *px{&x}; // px is pointer to x
                   // as a const int

// what are deduced types for param and T?
f(&x);  // T: ???, param: ???
f(px); // T: ???, param: ???
```

ParamType: Pointer/Reference (8/8)

- ParamType's type is T const*
- Ignore reference in expr and then pattern-match expr's type against ParamType to determine T

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

int x{27};          // x is an int
int const *px{&x}; // px is pointer to x
                    // as a const int

// what are deduced types for param and T?
f(&x);  // T: int, param: int const*
f(px); // T: int, param: int const*
```

ParamType: Neither Pointer Nor Reference (1/2)

- ParamType's type is T
- Fact that param is newly constructed object motivates rules governing how T is deduced from expr:
  - If expr's type is reference, ignore reference part
  - If expr is now const (or volatile), ignore that too

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

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to const int
int const * const rprx{&x};
// what are deduced types for param and T?
f(x);     // T: ???, param: ???
f(cx);    // T: ???, param: ???
f(rx);    // T: ???, param: ???
f(rprx);  // T: ???, param: ??
```

ParamType: Neither Pointer Nor Reference (2/2)

- ParamType's type is T
- Fact that param is new object motivates rules governing how T is deduced from expr:
  - If expr's type is reference, ignore reference part
  - If expr is now const (or volatile), ignore that too

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

int x{27};          // x is an int
int const cx{x};    // cx is const int
int const& rx{x};  // rx is reference to const int
int const * const rprx{&x};
// what are deduced types for param and T?
f(x);     // T: int, param: int
f(cx);    // T: int, param: int
f(rx);    // T: int, param: int
f(rprx); // T: int const*, param: int const*
```

ParamType: Forwarding Reference

- ParamType's type is T&&
- Situation is bit complicated because expr can be lvalue or rvalue expression!!!

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

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

Forwarding References (1/7)

- T&& means rvalue reference to some type T
- However, T&& has two different meanings
  - One meaning is rvalue reference
  - 2nd meaning is either lvalue reference or rvalue reference

```
void f(Widget&& param);     // rvalue reference
Widget&& var1 = Widget();   // rvalue reference
auto&& var2 = var1;         // not rvalue reference
template <typename T>
void f(std::vector<T>&& param); // rvalue reference
template <typename T>
void f(T&& param);  // not rvalue reference
```

Forwarding References (2/7)

- If you see T&& without type deduction, you're looking at rvalue references

```
void f(Widget&& param);   // rvalue reference
                          // no type deduction
Widget&& var1 = Widget(); // rvalue reference
                          // no type deduction
auto&& var2 = var1;       // not rvalue reference

template <typename T>
void f(std::vector<T>&& param); // rvalue reference
                                // no type deduction
template <typename T>
void f(T&& param);  // not rvalue reference
```

Forwarding References (3/7)

- Forwarding references arise in context of function template parameters
- In both cases below, template type deduction is taking place

```
template <typename T>
void f(T&& param);  // not rvalue reference

auto&& var2 = var1; // not rvalue reference
```

Forwarding References (4/7)

- Because forwarding references are references, they must be initialized
- Initializer determines whether lvalue or rvalue reference

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

Widget w;
f(w);     // lvalue passed to f
          // param's type is Widget&

f(std::move(w)); // rvalue passed to f
                 // param's type is Widget&&
```

Forwarding References (5/7)

- In addition to type deduction, form of reference declaration must be precisely T&& for a reference to be forwarding

```
template <typename T>
void f(std::vector<T>&& param); // rvalue reference
                                // form of param is not T&&
template <typename T>
void f(T const&& param); // rvalue reference
                  // presence of const is disqualifying

template <typename T>
void f(T&& param);  // not rvalue reference
```

Forwarding References (6/7)

- Being in a template doesn't guarantee type deduction

```
template <typename T,
          class Allocator = std::allocator<T>>
class vector {
public:
  void push_back(T&& x);
  ...
}
```

Forwarding References (7/7)

- Here, type parameter `Params` is independent of `vector`'s type parameter `T`, so `Params` must be deduced each time `emplace_back` is called

```
template<typename T,
         class Allocator = std::allocator<T>>
class vector {
public:
  template <class... Params>
  void emplace_back(Params&&... params);
  ...
};
```

ParamType: Forwarding Reference (1/3)

- ParamType's type is T&&
- When f is called with expr being an:
  - lvalue of type A, then T resolves to A&, and by reference collapsing rules, param's type is A&
  - rvalue of type A, then T resolves to A, and hence param's type is A&&
- ParamType is called *forwarding reference*

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

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

ParamType: Forwarding Reference (2/3)


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

int x{27};            // x is int
int const cx{x};  // cx is const int
int const& rx{x}; // rx is reference to int const

f(x);  // x:  ??? => T: ???, param: ???
f(cx); // cx: ??? => T: ???, param: ???
f(rx); // rx: ??? => T: ???, param: ???
f(27); // 27: ??? => T: ???, param: ???
```

ParamType: Forwarding Reference (3/3)


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

int x{27};          // x is int
int const cx{x};    // cx is const int
int const& rx{x};   // rx is reference to int const

f(x);  // x:  lvalue => T: int&, param: int&
f(cx); // cx: lvalue => T: int const&, param: int const&
f(rx); // rx: lvalue => T: int const&, param: int const&
f(27); // 27: rvalue => T: int, param: int&&
```

Type Deduction: Array Arguments (1/3)

- Array types are different from pointer types – even though they seem interchangeable
- Array *decays* into pointer to its first element:

```
char const name[] = "Clint";

// array decays to pointer
char const *ptr{name};
```

Type Deduction: Array Arguments (2/3)

- What happens if array is passed to template taking by-value parameter?

```
template <typename T>
void f(T param); // param is passed by value

char const name[] = "Clint";

// what type deduced for T and param?
f(name);
```

Type Deduction: Array Arguments (3/3)

- Although functions can’t declare parameters that are arrays, they can declare parameters that are references to arrays!

```
template <typename T>
void f(T& param); // param is passed by reference

char const name[] = "Clint";

// what type deduced for T and param?
f(name);
```

Deducing Array Size

- Ability to declare references to arrays enables creation of a template that deduces number of elements that an array contains:

```
// return array size as compile-time constant
template <typename T, std::size_t N>
constexpr std::size_t array_size(T (&)[N]) noexcept {
  return N;
}

int keys[] {1,3,5,7,9};

// vals has size 7
std::array<int, array_size(keys)> vals;
```

Type Deduction: Function Arguments

- Just like arrays, functions also decay into function pointers
- Type deduction is similar to arrays

```cpp
void func(int, double);

template <typename T> void f1(T param);

template <typename T> void f2(T& param);

// what is type of T and param?
f1(func);
// what is type of T and param?
f2(func);
```

auto Type Deduction (1/11)

- auto type deduction *is* template type deduction
- There is direct algorithmic transformation from template type deduction to auto 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;
```

`auto` Type Deduction (2/11)

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

`auto` Type Deduction (3/11)

- 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&& uref1 = x;     // uref1: ???
auto&& uref2 = cx;    // uref2: ???
auto&& uref3 = 27;    // uref3: ???
```

auto Type Deduction (4/11)


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

// functions
void  func(int, double); // arr1: ???
auto  fun1 = func;              // fun1: ???
auto& fun2 = func;              // fun2: ???
```

auto Type Deduction (5/11)

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

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

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

auto Type Deduction (6/11)

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

```
auto x1 = 27;   // type is int
auto x2(27);    // type is int
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 (7/11)

- When corresponding template is pass same initializer, type deduction fails

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

// template with parameter declaration
// equivalent to x's declaration
template <typename T>
void f(T param);

// error: cannot deduce type for T
f({11, 23, 9});
```

auto Type Deduction (8/11)

- This works though!!!

```
// template with parameter declaration
// equivalent to x's declaration
template <typename T>
void f(std::initializer_list<T> param);

// T deduced as int and param's type
// is std::initializer_list<int>
f({11, 23, 9});
```

auto Type Deduction (9/11)

- 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`

auto Type Deduction (10/11)

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

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

// error: cannot deduce type for {1,2,3}
auto create_initlist() {
  return {1, 2, 3}
}
```

auto Type Deduction (11/11)

- C++14 permits auto in lambda's parameter to be deduced using template type deduction

```
std::vector<int> v{1,2,3}, v2{11,22,33};
auto reset_vec =
  [&v](auto const& new_val) {v = new_val;};

// ok
reset_vec(v2);
// error: cannot deduce type for {1,2,3}
reset_vec({11, 22, 33});
```

auto Type Specifier: Summary

- 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 or lambda parameter implies template type deduction, not auto type deduction

decltype Type Specifier (1/8)

- decltype lets you find type of expression (without evaluating expression)

```
template <typename T>
auto incr(T x) {
  return x+1;
}

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

decltype Type Specifier (2/8)


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

decltype(i)      j{0};    // j has type ???
decltype(ri)     rj{j};   // rj has type ???
decltype(ci)     x{0};    // x has type ???
decltype(cj)     y{x};    // y has type ???
decltype(ci+10)  z{11};   // z has type ???
decltype(cj)     xx(10);  // xx has type ???
decltype(ri+10)  yy;      // yy has type ???
decltype(*pi)    zz{i};   // zz has type ???
```

decltype Type Specifier (3/8)

- One application of decltype is to declare function return types

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

decltype Type Specifier (4/8)

- One application of decltype is to declare function return types

```
// use same syntax as lambdas
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 Type Specifier (5/8)

- One application of decltype is to declare function return types

```
// 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 Type Specifier (6/8)

- One application of `decltype` is to declare function return types
- Since C++14, we can do this:

```
// auto specifes 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` Type Specifier (7/8)

- Want this to work for lvalues and rvalues

```
// auto speciifes 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 std::forward<Cont>(c)[i];
}
```

decltype Type Specifier (8/8)

- decltype(auto) not limited to function return types

```
int i;
int const &ci {i};

// auto type deduction
auto i1 = ci;               // i1 has type ???

// decltype type deduction
decltype(auto) i2 = ci; // i2 has type ???
```

decltype Type Specifier: Edge Cases (1/2)

- Applying decltype to a name yields declared type (lvalue expression) for that name
- For lvalue expressions more complicated than names (such as wrapping name in parentheses), decltype ensures that type reported is always lvalue reference

```
int i{0};
decltype(i) jj{i};  // jj has type ??
decltype((i)) jjj{0}; // jjj has type ???
```

decltype Type Specifier: Edge Cases (2/2)

- Something to worry about since C++14:

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

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

`decltype` Type Specifier: Summary

- `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