Type Deduction in C++14

Type Deduction in C++14

by David Stone
david@doublewise.net

Type deduction

- Type deduction has existed since C
- Expanded with every version of the language
- Remove redundancy
- Improve safety
- Efficiency

Examples of type deduction

- Temporary variables
  - f() + g() + h()

Examples of type deduction

- Converting (non-explicit) constructors
- Implicit conversion operators
- Overloaded functions
- f(x)

Examples of type deduction

- template parameters
- auto
- decltype

Examples of type computation

- meta-functions like std::common_type_t

Value category: the type of types

- prvalue == std::string
  - Construct a temporary, return by value
- lvalue == std::string &, std::string const &
  - Use a named variable, return by lvalue reference
- xvalue == std::string &&
  - Only when returned from a function...
- lvalue == std::string &&
  - Used as a named variable, function parameters

auto

- auto x = blah();
  - x is never a reference
  - Can be a pointer
  - Can be a std::reference_wrapper

auto

- auto && x = blah();
  - x is always a reference
  - const is deduced
  - Accepts anything
  - Turns "by value" (prvalue) into rvalue reference

auto

- auto & x = blah();
  - Always an lvalue-reference
  - const is deduced
  - Will not bind to an rvalue

auto

- auto const & x = blah();
  - Always an lvalue-reference
  - Accepts anything
    - Will bind to an rvalue and extend lifetime

decltype(auto)

- decltype(auto) x = blah();
  - x is a reference if blah returns by reference
    - value category is deduced
  - const is deduced

Summary

- auto is type deduction by value
- auto && is type deduction by reference
- Same as template arguments
  - T vs. T &&
- decltype(auto) is type deduction + reference deduction
  - That bullet point is a lie

Redundancy

```
char const * ptr = reinterpret_cast<char const *>(&x);


std::vector<int> make_vector(std::size_t size) {
    return std::vector<int>(size);
}
```

Redundancy

- int * ptr = malloc(sizeof(int));
  - C version
- int * ptr = malloc(sizeof(long));
  - Works on Windows!
- int * ptr = static_cast<int *>(malloc(sizeof(int)));
  - "Bad" C++98
- int * ptr = new int;
  - Better
- auto ptr = std::make_unique<int>();

Don't Repeat Yourself (DRY)

- Good type deduction lets you say what you mean exactly once
- With type deduction, we can build powerful, reusable abstractions

Powerful abstractions

- If one feature can do everything another feature can, plus more, the weaker feature is useless
  - Why have two ways to do the same thing?

Casts

- C-style cast
  - static_cast
  - const_cast
  - reinterpret_cast
  - And more!
- Yet we still have (and want) the C++ casts
- Safety

push_back vs. emplace_back

- push_back will copy or move the object
- emplace_back constructs the object in place
  - Accepts any number of arguments and forwards
- v.push_back(Thing(5, "pizza"));
- v.emplace_back(5, "pizza");
- v.push_back({5, "pizza"});
- What if there is only one argument?

push_back vs. emplace_back

- push_back will only call implicit constructors
- emplace_back will call explicit constructors

push_back vs. emplace_back

```
std::vector<std::unique_ptr<int>> v;
int a = 5;
v.emplace_back(&a); // compiles
v.push_back(&a); // does not compile
```

Redundancy

```
useful_type_information value = some_function();
auto value = some_function();

std::vector<Customer> v = some_function();
auto customers = some_function();

double thrust = calculate_thrust();
```

Why

- What it means vs. what you want it to mean
- `int16_t BH = calculate_horizontal_bias();`
- `auto BH = cast<int16_t>(calculate_horizontal_bias());`
- `auto BH = calculate_horizontal_bias();`

Return type deduction

- auto f() { return 5; }
- decltype(auto) g() { return 5; }
- auto && h() { return 5; } // dangling reference

Return type deduction: question

- auto a() { return "a"; }
  - char const *
  - char const (&)[2]
  - something else

Return type deduction: answer

- auto a() { return "a"; }
  - char const *
- Arrays cannot be passed by value
  - Decay to pointer to first element

Return type deduction: question

- auto && b() { return "b"; }
  - char const *
  - char const (&)[2]
  - something else

Return type deduction: answer

- auto && b() { return "b"; }

  - char const (&)[2]

Return type deduction: question

- decltype(auto) c() { return "c"; }
  - char const *
  - char const (&)[2]
  - something else

Return type deduction: answer

- decltype(auto) c() { return "c"; }
  - char const (&)[2]
- decltype(auto) is the type as declared

Strings

```
struct S {
    S(std::string const & s): data(s){}


    std::string const & data;
};
auto const hi = "Hi";
S s(hi);
S s2("Hi");
```

Deleting bad overloads

```
struct S {
    S(std::string const & s): data(s){}
    S(std::string &&) = delete;
    std::string const & data;
};
auto const hi = "Hi";
S s(hi);
S s2("Hi");
```

Deleting bad overloads

```
template<typename T>
S(T && s): data(s) {
    static_assert(
        std::is_lvalue_reference<T &&>::value,
        "No temporaries."
    );
}
```

Member functions

- struct S { std::string m; };
- auto S::a() { return m; }
  - std::string
- auto && S::b() { return m; }
  - std::string &

Member functions, ref qualifiers

- struct S { std::string m; };
- decltype(auto) S::f() const & { return m; }
  - std::string
- decltype(auto) S::g() & { return m; }
  - std::string
- decltype(auto) S::h() && { return m; }
  - std::string

Turn a variable into an expression

- S & f();
- decltype(f().m) == std::string
- decltype((f().m)) == std::string &

Turn a variable into an expression

- S const & f();
- decltype(f().m) == std::string
- decltype((f().m)) == std::string const &

Turn a variable into an expression

- S && f();
- decltype(f().m) == std::string
- decltype((f().m)) == std::string &&

Turn a variable into an expression

- S f();
- decltype(f().m) == std::string
- decltype((f().m)) == std::string

Member functions, ref qualifiers

- struct S { std::string m; };
- auto && S::c() const & { return m; }
  - std::string const &
- auto && S::d() & { return m; }
  - std::string &
- auto && S::e() && { return m; }
  - std::string &

The cautious committee

```
string f(string s) { return s; }
string g(string && s) { return s; }


unique_ptr<int> f(unique_ptr<int> p) { return p; }
unique_ptr<int> g(unique_ptr<int> && p) { return p; }
```

Member functions, ref qualifiers

- struct S { std::string m; };
- auto && S::e() && { return std::move(m); }
  - std::string &&

Forwarding function parameters

```
template<typename T>
decltype(auto) parameter(T && x) {
    return std::forward<T>(x);
}
template<typename T>
auto && parameter(T && x) {
    return std::forward<T>(x);
}
```

Forwarding function parameters

```
template<typename T>
decltype(auto) variable(T && x) {
    return (std::forward<T>(x).member);
}
template<typename T>
auto && variable(T && x) {
    return std::forward<T>(x).member;
}
```

Forwarding function parameters

```
template<typename T>
decltype(auto) member(T && x) {
    return std::forward<T>(x).member();
}
```

Constructors

```
struct S {
    S() = default;


    template<typename T>
    S(T && x) { cout << "Hi there!\n"; }
};
S s1;
S s2(s1);
```

Constructors

```
struct S {
    S() = default;
    S(S const &) = default;
    template<typename T>
    S(T && x) { cout << "Hi there!\n"; }
};
S s1;
S s2(s1);
```

Constructors

```
struct S {
    S() = default;
    S(S const &) = default;
    template<typename T = S &>
    S(T && x) { cout << "Hi there!\n"; }
};
S s1;
S s2(s1);
```

Constructors

- T && can "hide" copy constructor when T is deduced as S & instead of S const &
- S & && => S &
  - "Reference collapsing"

Deduced context

- auto &&
  - Deduced
- template<typename T> void f(T &&)
  - Deduced
- template<typename T> void g(std::vector<T> &&)
  - Not deduced
  - Rvalue reference only

std::array vs. C-arrays

- std::array<int, 4> a{ 2, 3, 5, 7 };
- int b[4]{ 2, 3, 5, 7 };
- int c[]{ 2, 3, 5, 7 };
- C code has more type deduction than C++?

make_array

```
template<typename T, typename... Ts>
constexpr auto make_array(Ts && ... args) {
    array<T, sizeof...(Ts)>{
        forward<Ts>(args)...
    };
}
auto array = make_array<int>(1, 3, 3, 7);
```

Multi-dimensional array

```
int d[][2] { { 2, 7 }, { 1, 8 } };
auto e = make_array<array<int, 2>>(
    make_array(2, 7),
    make_array(1, 8)
);
```

Ideal multi-dimensional make_array

```
auto cool = make_array<string, 3>(
    "Hero", "of", "Canton",
    "man", "called", "Jayne"
);

cool == array<array<string, 3>, 2>
```

multi_dimensional_array

```
template<typename T, size_t dim, size_t... dims>
struct multi_dimensional_array {
    using type = array<typename multi_dimensional_array<T, dims...>::type, dim>;
};
template<typename T, size_t dim>
struct multi_dimensional_array<T, dim> {
    using type = array<T, dim>;
};
```

multi_dimensional_array

```
template<typename T, size_t... dimensions>
using multi_dimensional_array_t = typename
multi_dimensional_array<T, dimensions...>::type;


// alias template, "template typedef"
```

multi_dimensional_array

```
array<array<array<int, 5>, 4>, 3>

typename multi_dimensional_array<int, 3, 4, 5>::type

multi_dimensional_array_t<int, 3, 4, 5>

int [3][4][5]

array<int, 3, 4, 5>
```

make_array

```
template<typename T, size_t... dims, typename... Ts>
constexpr auto make_array(Ts && ... args) {
    return multi_dimensional_array_t<
        T,
        final_dimension<sizeof...(Ts), dims...>::value,
        dims...
    >{ std::forward<Ts>(args)... };
}
```

The final dimension

```
template<size_t n, size_t... dims>
struct final_dimension {
    static_assert(
        n % product<dims...> == 0,
        "Unable to deduce final dimension."
    );
    static constexpr size_t value = n / product<dims...>;
};
```

Variadic product

```
template<size_t... xs>
constexpr size_t product = 1;
template<size_t x, size_t... xs>
constexpr size_t product<x, xs...> = x * product<xs...>;
```

The final dimension (C++17)

```
template<size_t n, size_t... dims>
struct final_dimension {
    static_assert(
        n % (dims * ...) == 0,
        "Unable to deduce final dimension."
    );
    static constexpr size_t value = n / (dims * ...);
};
```

make_array

```
template<typename T, size_t... dims, typename... Ts>
constexpr auto make_array(Ts && ... args) {
    return multi_dimensional_array_t<
        T,
        final_dimension<sizeof...(Ts), dims...>::value,
        dims...
    >{ std::forward<Ts>(args)... };
}
```

“As good as C!”

- Not exactly a high benchmark
- Can we do better?

std::common_type

- Metafunction
  - Function-like thing that operates on types instead of values
- Returns a type that all arguments can be implicitly converted to

std::common_type

- common_type_t<int, long>
  - long
- common_type_t<int const, int>
  - int
- common_type_t<int volatile &, int volatile &>
  - int
- Arguments are decayed
  - Kind of...

Deduce the type

```
template<size_t... dims, typename... Ts>
constexpr auto make_array(Ts && ... args) {
    return multi_dimensional_array<
        std::common_type_t<std::decay_t<Ts>...>,
        final_dimension<sizeof...(Ts), dims...>::value,
        dims...
    >{ std::forward<Ts>(args)... };
}
```

Overloading function templates

```
template<typename T>
bool f() { return true; }

template<int n>
string f() { return "Ahhhhhh!"; }

f<long>();
f<7>();
```

Implementation of common_type

```
template<typename T, typename U>
struct common_type {
    using type = decay_t<decltype(true ?
        std::declval<T>() :
        std::declval<U>()
    )>;
};
```

Specializations of common_type

```
template<>
struct common_type<MyType, MyOtherType> {
    using type = MoreGeneralType;
};
```

Templates do not convert

```
template<typename T, typename U>
auto f(T &&, U &&) {
    return std::common_type_t<T, U>{};
}
```

bounded::integer

```
constexpr bounded::integer<0, 10> a = thing1();
constexpr bounded::integer<5, 7> b = thing2();
constexpr auto c = a + b;
// decltype(c) == bounded::integer<5, 17>
constexpr bounded::integer<0, 4> d = c; // compile error
// assume c == 10
constexpr bounded::integer<0, 9> e(d); // compile error
```

bounded::integer overflow

- integer<-128, 127>
  - integer<-128, 127, null_policy>
- checked_integer<15, 32>
  - integer<15, 32, throw_policy>
- clamped_integer<1, 100>
  - integer<1, 100, clamp_policy>
- wrapping_integer<0, 255>
  - integer<0, 255, modulo_policy>

common_type of a bounded::integer

- What does common_type mean?
  - The type that both types can convert to
- using T = bounded::integer<0, 10>;
- using U = bounded::integer<5, 15>;
- common_type_t<T, U> == bounded::integer<0, 15>

Using bounded::integer

```
auto calculate_something_safely() {
    bounded::checked_integer<0, 100> const x = read_from_file();
    return (x +20_bi) / 30_bi;
}
auto result = calculate_something_safely();
static_assert(std::is_same<result, bounded::checked_integer<0, 4>>::value, "Unexpected type.");
```

Naming a type: compile-time printing

- template<typename T> class Print;
  - Note, the class is not defined
  - Not template<typename T> class Print{};
- Print<decltype(expression)> p;
  - Error message about using undefined type
  - Has your type name in it

Checking if a nested type exists

```
using yes = char[1];
using no = char[2];
#define HAS_MEMBER_TYPE(type) \
\
template<typename T> \
auto checker_ ## type(typename T::type *) -> yes &; \
\
template<typename> \
auto checker_ ## type(...) -> no &;
```

Checking if a nested type exists

```
HAS_MEMBER_TYPE(MyTag)


template<typename T>
struct is_overflow_policy : std::integral_constant<
    bool,
    sizeof(checker_MyTag<T>(nullptr)) == sizeof(yes)
>{};
```