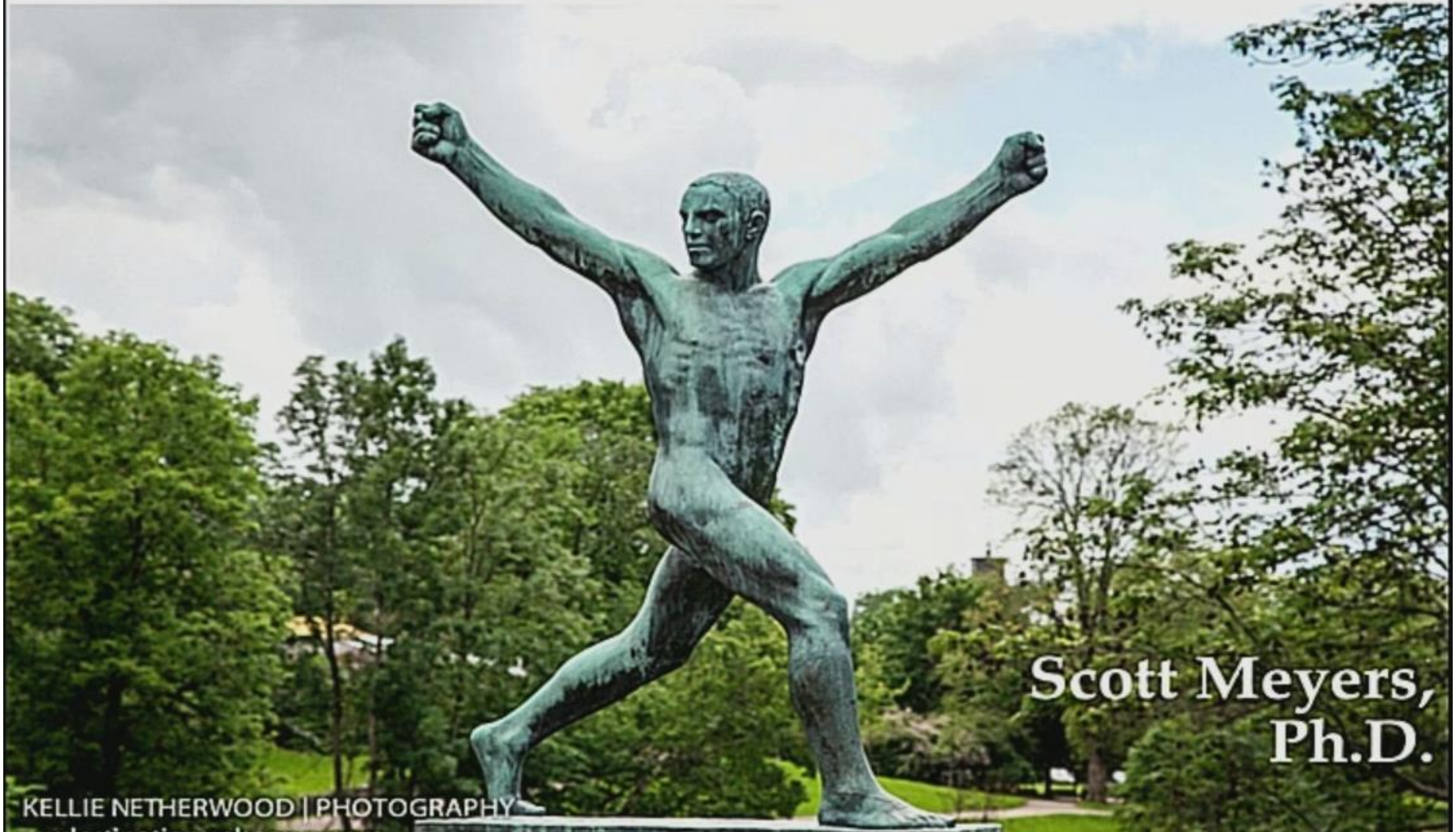
C++ Type Deduction and Why You Care



Why You Care

It Just Works

In C++98, type deduction used only for templates.

- Generally *just works*.
- Detailed understanding rarely needed.

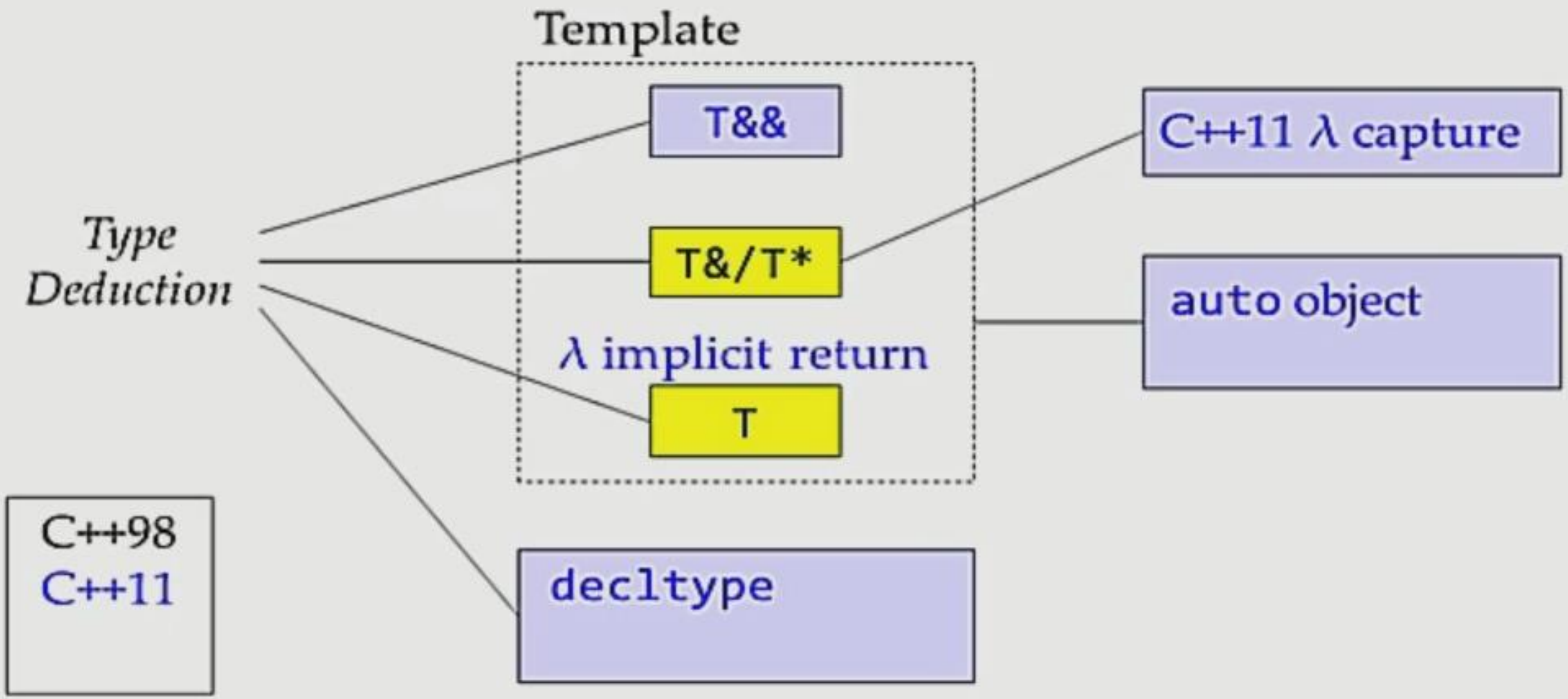
In C++11, scope expands:

- `auto` variables, universal references, lambda captures and returns, `decltype`.
- *Just works* less frequently.
  - ➡ Six sets of rules!

In C++14, scope expands further:

- Function return types, lambda init captures.
- Same rulesets, but more usage contexts (and chances for confusion).

The C++ Type Deduction Landscape



Template Type Deduction

General problem:

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

f(expr);                    // deduce T and ParamType from expr
```

Given type of `expr`, what are these types?

- `T`
  - ➡ The deduced type.
- *`ParamType`*
  - ➡ Often different from `T` (e.g, `const T&`).

Three general cases:

- *`ParamType`* is a reference or pointer, but not a universal reference.
- *`ParamType`* is a universal reference.
- *`ParamType`* is neither reference nor pointer.

Non-URef Reference/Pointer Parameters

Type deduction very simple:

- If *expr*'s type is a reference, ignore that.
- Pattern-match `expr`'s type against `ParamType` to determine `T`.

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

int x = 22;                // int
const int cx = x;          // copy of int
const int& rx = x;         // ref to const view of int

f(x);                      // T ≡ int, param's type ≡ int&

f(cx);                     // T ≡ const int,
                           // param's type ≡ const int&

f(rx);                     // T ≡ const int,
                           // param's type ≡ const int&
```

➡ Note: `T` not a reference.

Non-URef Reference/Pointer Parameters

ParamType of `const T&` ⇒ `T` changes, but `param`'s type doesn't:

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

int x = 22;                     // as before
const int cx = x;               // as before
const int& rx = x;              // as before

f(x);                           // T ≡ int, param's type ≡ const int&

f(cx);                          // T ≡ const int,
                                // param's type ≡ const int&

f(rx);                          // T ≡ const int,
                                // param's type ≡ const int&
```

➡ Note: `T` not a reference.

Non-URef Reference/Pointer Parameters

Behavior with pointers essentially the same:

```
template<typename T>
void f(T* param);           // param now a pointer

int x = 22;                 // int
const int *px = &x;         // ptr to const view of int

f(&x);                      // T ≡ int, param's type ≡ int*

f(px);                      // T ≡ const int,
                            // param's type ≡ const int*
```

- Note: T not a pointer.

Behavior of `const T*` parameters as you'd expect.

auto and Non-URef Reference/Pointer Variables

auto plays role of T:

```
int x = 22;                 // as before
const int cx = x;           // as before
const int& rx = x;          // as before

auto& v1 = x;               // v1's type ≡ int& (auto ≡ int)

auto& v2 = cx;              // v2's type ≡ const int&
                            // (auto ≡ const int)

auto& v3 = rx;              // v3's type ≡ const int&
                            // (auto ≡ const int)


const auto& v4 = x;         // v4's type ≡ const int& (auto ≡ int)

const auto& v5 = cx;        // v5's type ≡ const int&
                            // (auto ≡ const int)

const auto& v6 = rx;        // v6's type ≡ const int&
                            // (auto ≡ const int)
```

Yawn

Type deduction for non-URef reference/pointer parameters/variables quite intuitive.

Universal References

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

f(expr);
```

Treated like "normal" reference parameters, except:

- If *expr* is lvalue with deduced type `E`, `T` deduced as `E&`.
  - ➡ Reference-collapsing yields type `E&` for `param`.

```
int x = 22;                  // as before
const int cx = x;            // as before
const int& rx = x;           // as before

f(x);         // x is lvalue ⇒ T ≡ int&, param's type ≡ int&

f(cx);        // cx is lvalue ⇒ T ≡ const int&,
              // param's type ≡ const int&

f(rx);        // rx is lvalue ⇒ T ≡ const int&,
              // param's type ≡ const int&

f(22);        // x is rvalue ⇒ no special handling;
              // T ≡ int, param's type is int&&
```

By-Value Parameters

Deduction rules a bit different (vis-à-vis by-reference/by-pointer):

- As before, if *expr*'s type is a reference, ignore that.
- If *expr* is `const` or `volatile`, ignore that.
- `T` is the result.

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

int x = 22;                   // as before
const int cx = x;             // as before
const int& rx = x;            // as before
f(x);                         // T ≡ int, param's type ≡ int
f(cx);                        // T ≡ int, param's type ≡ int
f(rx);                        // T ≡ int, param's type ≡ int
```

expr's reference-/`const`-qualifiers always dropped in deducing `T`.

Non-Reference Non-Pointer autos

`auto` again plays role of `T`:

```
int x = 22;                // as before
const int cx = x;          // as before
const int& rx = x;         // as before

auto v1 = x;               // v1's type ≡ int (auto ≡ int)

auto v2 = cx;              // v2's type ≡ int (auto ≡ int)

auto v3 = rx;              // v3's type ≡ int (auto ≡ int)
```

Again, *expr*'s reference-/`const`-qualifiers always dropped in deducing `T`.

- `auto` never deduced to be a reference.
  - ➡ It must be manually added.
    - ◆ If present, use by-reference rulesets.

```
auto v4 = rx;              // v4's type ≡ int

auto& v5 = rx;             // v5's type ≡ const int&

auto&& v6 = rx;            // v6's type ≡ const int&
                           // (rx is lvalue)
```

const *exprs* vs. *exprs* Containing const

Consider:

```
void someFunc(const int * const param1,   // const ptr to const
              const int *       param2,   // ptr to const
                    int *       param3)   // ptr to non-const
{
  auto p1 = param1;     // p1's type ≡ const int*
                        // (param1's constness ignored)
  auto p2 = param2;     // p2's type ≡ const int*
  auto p3 = param3;     // p3's type ≡ int*
  …
}
```

From earlier:

- If *expr* is `const` or `volatile`, ignore that.

More common wording:

- *Top-level* `const`/`volatile` is ignored.

const *exprs* vs. *exprs* Containing const

Applies only when deducing types for non-reference non-pointer parameters/variables:

```
void someFunc(const int * const param1,     // as before
              const int *       param2,     // as before
                    int *       param3)     // as before
{
  auto p1 = param1;            // p1's type ≡ const int*
                               // (param1's constness ignored)

  auto& p2 = param1;           // p2's type ≡ const int * const&
                               // (param1's constness not ignored)

  …
}
```

Special Cases

Special treatment for `exprs` that are arrays or functions:

- When initializing a reference, array/function type deduced.
- Otherwise *decays* to a pointer before type deduction.

auto Type Deduction

Same as template type deduction, except with braced initializers.

- Template type deduction fails.
- auto deduces `std::initializer_list`.

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

f( { 1, 2, 3 } );              // error! type deduction fails

auto x = { 1, 2, 3 };          // x's type ≡
                               //   std::initializer_list<int>
```

Observing Deduced Types

Use declared-only template with type of interest:

```
template<typename T>                    // declaration for TD;
class TD;                               // TD == "Type Displayer"

template<typename T>                    // template w/types
void f(T& param)                        // of interest
{
  TD<T> tType;                          // cause T to be shown
  TD<decltype(param)> paramType;        // ditto for param's type
  …
}
```

Observing Deduced Types

```
int x = 22;                 // as before
const int& rx = x;          // as before

f(rx);                      // compiler diagnostics show types
```

gcc 4.8 (excerpt):

```
error: 'TD<const int> tType' has incomplete type

error: 'TD<const int &> paramType' has incomplete type
```

VS 2013 (excerpt):

```
error C2079: 'tType' uses undefined class 'TD<T>'
        with
        [
            T=const int
        ]
error C2079: 'paramType' uses undefined class 'TD<T &>'
        with
        [
            T=const int
        ]
```

Observing Deduced Types

Clang 3.2 (excerpt):

```
error: implicit instantiation of undefined template 'TD<const int>'
error: implicit instantiation of undefined template 'TD<const int &>'
```

Observing Deduced Types

For `auto` variables, use `decltype` to get type:

```
int x = 22;                    // as before
const int& rx = x;             // as before

auto y = rx;
TD<decltype(y)> yType;         // compiler diagnostics show type
```

gcc 4.8 (excerpt):

```
error: aggregate 'TD<int> yType' has incomplete type and
    cannot be defined
```

VS 2013 (excerpt):

```
error C2079: 'yType' uses undefined class 'TD<int>'
```

Clang 3.2 (excerpt):

```
error: implicit instantiation of undefined template 'TD<int>'
```

decltype Type Deduction

`decltype(`*`name`*`)` ≡ declared type of *name*. Unlike `auto`:

- Never strips `const`/`volatile`/references.

```
int x = 10;                    // decltype(x) ≡ int
const auto& rx = x;            // decltype(rx) ≡ const int&
```

decltype Type Deduction

decltype(*lvalue expr of type T*) ≡ T&.

- Unsurprising. Almost all such expressions really have type T&.

```
const std::vector<Widget>&               // return type is lvalue
  findVec(const VecHandle&);             // expr and has lvalue-ref
                                         // type
```

➡ Exceptions act as if they did.

```
int arr[10];

arr[0] = 5;                              // arr[0]'s type is int,
                                         // but it acts like int&

decltype(arr[0]) ≡ int&                  // because arr[0] is
                                         // lvalue expression
```

`decltype` Type Deduction

Full rules for `decltype` more complex.

- Relevant only to hard-core library developers.
- Rules we've seen suffice for almost everybody almost all the time.

Names as Lvalue Expressions

Names are lvalues, but `decltype(`*`name`*`)` rule beats `decltype(`*`expr`*`)` rule:

```
int x;

decltype(x) ≡ int              // x is lvalue expression, but
                               // also a name ⇒ name rule prevails

decltype((x)) ≡ int&           // (x) is lvalue expression, but
                               // isn't a name
```

Implication of "superfluous parentheses" apparent soon.

Function Return Type Deduction

In C++11:

- **Limited**: single-statement lambdas only.

In C++14:

- **Extensive**: all lambdas + all functions.
  ➡ Understanding type deduction more important than ever.

Deduced return type specifiers:

- `auto`: Use template (not `auto`!) type deduction rules.
  ➡ No type deduced for braced initializers.
- `decltype(auto)`: Use `decltype` type deduction rules.

Function Return Type Deduction

Sometimes `auto` is correct:

```
auto lookupValue( context information )
{
  static std::vector<int> values = initValues();
  int idx = compute index into values from context info;
  return values[idx];
}
```

- Returns `int`.
- `decltype(auto)` would return `int&`.
  - ➡ Would permit caller to modify `values`!

```
lookupValue(myContextInfo) = 0;        // shouldn't compile!
```

Function Return Type Deduction

Sometimes `decltype(auto)` is correct:

```
decltype(auto) authorizeAndIndex(std::vector<int>& v, int idx)
{
  authorizeUser();
  return v[idx];
}
```

- Returns `int&`.
- `auto` would return `int`.
  - ➡ Wouldn't permit caller to modify `std::vector`:

```
    authorizeAndIndex(myVec, 10) = 0;        // should compile!
```

Function Return Type Deduction

`decltype(auto)` sensitive to function implementation:

```
decltype(auto) lookupValue( context information )
{
  static std::vector<int> values = initValues();
  int idx = compute index into values from context info;

  auto retVal = values[idx];        // retVal's type is int

  return retVal;                    // returns int
}

decltype(auto) lookupValue( context information )
{
  static std::vector<int> values = initValues();
  int idx = compute index into values from context info;

  auto retVal = values[idx];        // retVal's type is int

  return (retVal);                  // returns int& (to local
}                                   // variable!)
```

Function Return Type Deduction

Rules of thumb:

- Use `auto` if a reference type would never be correct.
- Use `decltype(auto)` only if a reference type could be correct.

Further Information

- "C++ auto and decltype Explained," Thomas Becker, `thbecker.net`, May 2013.
- "Capture Quirk in C++14," Scott Meyers, *The View From Aristeia* (blog), 3 February 2014.
- *Effective Modern C++*, Scott Meyers, O'Reilly, anticipated October 2014.