Effective Modern C++ Documentation

Release 0.0.1

Hans-J. Schmid

Apr 04, 2018

Contents:

CHAPTER 1

1. Deducing Types

1.1 Item 1: Understand template type deduction

Key idea:

If the function template looks like this:

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

then two types are deduced: one for `T` and one for `ParamType`. These types are often different, because `ParamType` can contain adornments, e.g. const or reference qualifiers.

Listing 1.1: 01-pinch_of_pseudocode.cpp

```
template <typename T>
void f(const T& param) {}  // ParamType is const T&

int main() {
  int x = 0;
  f(x);  // call f with an int
}
```

1.1.1 Case 1: ParamType is a Reference or Pointer, but not a Universal Reference

Key idea:

Considering the general form for templates and calls to it:

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

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

then, in the simplest case when `ParamType` is a reference type or a pointer type, but not a universal reference, type deduction works like this:

1. If expr's type is a reference, ignore the reference part.
2. Then pattern-match expr's type against `ParamType` to determine `T`.

Listing 1.2: 02-case1_non_const.cpp

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

int main() {
  int x = 27;          // x is an int
  const int cx = x;    // cx is a const int
  const int& rx = x;   // rx is a reference to x as a const int

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

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

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

Key idea:

Considering the general form for templates and calls to it:

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

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

then, in the simplest case when `ParamType` is a pointer type or a reference type, but not a universal reference, type deduction works like this:

1. If expr's type is a reference, ignore the reference part.
2. Then pattern-match expr's type against `ParamType` to determine `T`.

If the type of f's parameter is changed from `T&` to `const T&`, the constness of cx and rx continues to be respected, but because we're now assuming that param is a reference-to-const, there's no longer a need for const to be deduced as part of `T`.

Listing 1.3: 03-case1_const.cpp

```
template <typename T>
void f(const T& param) {}  // param is now a ref-to-const

int main() {
  int x = 27;          // as before
  const int cx = x;    // as before
  const int& rx = x;   // as before

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

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

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

Key idea:

Considering the general form for templates and calls to it:

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

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

then, in the simplest case when `ParamType` is a pointer type or a reference type, but not a universal reference, type deduction works like this:

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

If param were a pointer (or a pointer to const) instead of a reference, things would work essentially the same way.

Listing 1.4: 04-case1_pointer.cpp

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

int main() {
  int x = 27;           // as before
  const int* px = &x;  // px is a ptr to x as a const int

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

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

1.1.2 Case 2: ParamType is a Universal Reference

Key idea:

Considering the general form for templates and calls to it:

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

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

then, in the case when `ParamType` is a universal reference type, type deduction works like this:

- If expr is an lvalue, both `T` and `ParamType` are deduced to be lvalue references
- If expr is an rvalue, the usual type deduction rules apply.

Listing 1.5: 05-case2_uref.cpp

```
template <typename T>
void f(T&& param) {}  // param is now a universal reference
```

```
int main() {
  int x = 27;          // as before
  const int cx = x;    // as before
  const int& rx = x;   // as before

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

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

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

  f(27);  // 27 is rvalue, so T is int,
          // param's type is therefore int&&
}
```

1.1.3 Case 3: ParamType is Neither a Pointer nor a Reference

Key idea:

If we're dealing with pass-by-value

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

That means that param will be a copy of whatever is passed in - a completely new object. The fact that param will be a new object motivates the rules that govern how `T` is deduced from expr:

1. As before, if expr's type is a reference, ignore the reference part.
2. If, after ignoring expr's reference-ness, expr is const, ignore that, too. If it's volatile, also ignore that. (volatile objects are uncommon. They're generally used only for implementing device drivers.)

Listing 1.6: 06-case3_pass_by_value.cpp

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

int main() {
  int x = 27;          // as before
  const int cx = x;    // as before
  const int& rx = x;   // as before
  f(x);                // T's and param's types are both int

  f(cx);  // T's and param's types are again both int

  f(rx);  // T's and param's types are still both int

  const char* const ptr =  // ptr is const pointer to const object
      "Fun with pointers";

  f(ptr);  // pass arg of type const char * const
}
```

1.1.4 Array Arguments

Key idea:

> In many contexts, an array decays into a pointer to its first element.

Listing 1.7: 07-array-to-pointer_decay_rule.cpp

```
int main() {
  const char name[] = "J. P. Briggs";  // name's type is
                                       // const char[13]

  const char* ptrToName = name;  // array decays to pointer
}
```

Key idea:

> Because array parameter declarations are treated as if they were pointer parameters, the type of an array that's passed to a template function by value is deduced to be a pointer type.

Listing 1.8: 08-arrays_by_value.cpp

```
template <typename T>
void f(T param) {}  // template with by-value parameter

int main() {
  const char name[] = "J. P. Briggs";  // name's type is
                                       // const char[13]

  f(name);  // what types are deduced for T and param?
            // -> name is array, but T deduced as const char*
}
```

Key idea:

> Although functions can't declare parameters that are truly arrays, they can declare parameters that are references to arrays.
>
> *The type deduced for* `T` *is the actual type of the array!* That type includes the size of the array, so in this example `T` is deduced to be `const char[13]`, and the type of f's parameter (a reference to this array) is `const char (&)[13]`.

Listing 1.9: 09-arrays_by_reference.cpp

```
template <typename T>
void f(T& param) {}  // template with by-reference parameter

int main() {
  const char name[] = "J. P. Briggs";  // name's type is
                                       // const char[13]

  f(name);  // pass array to f
}
```

Key idea:

> The ability to declare references to arrays enables creation of a template to deduce the number of elements that an array contains.

Listing 1.10: 11-deduce_nb_array_elements.cpp

```
#include <array>
#include <cstddef>

// return size of an array as a compile-time constant. (The
// array parameter has no name, because we care only about
// the number of elements it contains.)
template <typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept {
  return N;
}

// keyVals has 7 elements
int keyVals[] = {1, 3, 7, 9, 11, 22, 35};

// so does mappedVals
int mappedVals1[arraySize(keyVals)];

// mappedVals' size is 7
std::array<int, arraySize(keyVals)> mappedVals2;
```

1.1.5 Function Arguments

Key-idea:

> Function types can decay into pointers, too, and everything regarding type deduction and arrays applies to type deduction for functions and their decay into function pointers.

Listing 1.11: 10-function-to-pointer_decay_rule.cpp

```
void someFunc(int, double) {}  // someFunc is a function;
                               // type is void(int, double)

template <typename T>
void f1(T param) {}  // in f1, param passed by value

template <typename T>
void f2(T& param) {}  // in f2, param passed by ref

int main() {
  f1(someFunc);  // param deduced as ptr-to-func;
                 // type is void (*)(int, double)

  f2(someFunc);  // param deduced as ref-to-func;
                 // type is void (&)(int, double)
}
```

> Identical function declarations.

Listing 1.12: 12-array_and_pointer_parameter_equivalence.cpp

```
void myFunc1(int param[]) {}

void myFunc2(int* param) {}  // same function as above
```

1.1.6 Things to Remember

- During template type deduction, arguments that are references are treated as non-references, i.e. their reference-ness is ignored.
- When deducing types for universal reference parameters, lvalue arguments get special treatment and are deduced as lvalue references. It's the only situation in template type deduction where T is deduced to be a reference
- When deducing types for by-value parameters, const or volatile arguments are treated as non-const and non-volatile.
- During template type deduction, arguments that are array or function names decay to pointers, unless they're used to initialize references.

1.2 Item 2: Understand auto type deduction

Key idea:

> Deducing types for auto is the same as deducing types for templates (with only one curious exception).

Listing 1.13: 1-auto_type_deduction.cpp

```
template <typename T>         // conceptual template for
void func_for_x(T param) {}  // deducing x's type

template <typename T>                // conceptual template for
void func_for_cx(const T param) {}  // deducing cx's type

template <typename T>                 // conceptual template for
void func_for_rx(const T& param) {}  // deducing rx's type

void someFunc(int, double) {}  // someFunc is a function;
                               // type is void(int, double)

int main() {
  auto x = 27;  // case 3 (x is neither ptr nor reference)

  const auto cx = x;  // case 3 (cx isn't either)

  const auto& rx = x;  // case 1 (rx is a non-universal ref.)

  auto&& uref1 = x;  // x is int and lvalue,
                     // so uref1's type is int&

  auto&& uref2 = cx;  // cx is const int and lvalue
                      // so uref2's type is const int&

  auto&& uref3 = 27;  // 27 is int and rvalue,
                      // so uref3's type is int&&
```

```
  func_for_x(27);  // conceptual call: param's
                   // deduced type is x's type

  func_for_cx(x);  // conceptual call: param's
                   // deduced type is cx's type

  func_for_rx(x);  // conceptual call: param's
                   // deduced type is rx's type

  const char name[] =  // name's type is const char[13]
      "R. N. Briggs";

  auto arr1 = name;  // arr1's type is const char*

  auto& arr2 = name;  // arr2's type is
                      // const char (&)[13]

  auto func1 = someFunc;  // func1's type is
                          // void (*)(int, double)

  auto& func2 = someFunc;  // func2's type is
                           // void (&)(int, double)
}
```

Key idea:

> The treatment of braced initializers is the only way in which auto type deduction and template type deduction differ.

Listing 1.14: 2-auto_deduction_vs_template_deduction.cpp

```
#include <initializer_list>

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

template <typename T>
void f2(std::initializer_list<T> initList) {}

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

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

    // Error! Can't deduce T for std::initializer_list<T>
    // auto x5 = {1, 2, 3.0};
```

```
  }

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

    // Error! Can't deduce type for T
    // f({ 11, 23, 9 });

    // T deduced as int, and initList's type is std::initializer_list<int>
    f2({11, 23, 9});
  }
}
```

Key ideas:

1. A function with an auto return type that returns a braced initializer list won't compile.
2. When auto is used in a parameter type specification in a C++14 lambda expression, things won't compile.

Listing 1.15: 3-function_return_type_deduction.cpp

```
#include <vector>

auto createInitList() {
  // return {1, 2, 3};    // error: can't deduce type
  // for {1, 2, 3}
}

int main() {
  std::vector<int> v;

  auto resetV = [&v](const auto& newValue) { v = newValue; };  // C++14

  // Error! Can't deduce type for { 1, 2, 3 }
  // resetV( {1, 2, 3} );
}
```

1.2.1 Things to Remember

- auto type deduction is usually the same as template type deduction, but auto type deduction assumes that a braced initializer represents a `std::initializer_list`, and template type deduction doesn't.
- auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.

CHAPTER 2

2. auto

CHAPTER 3

3. Moving to Modern C++

CHAPTER 4

4. Smart Pointers

CHAPTER 5

5. Rvalue References, Move Semantics, and Perfect Forwarding

CHAPTER 6

6. Lambda Expressions

CHAPTER 7

7. The Concurrency API

CHAPTER 8

8. Tweaks

CHAPTER 9

Indices and tables

- genindex
- modindex
- search