Type deducing and auto type specifier

Yoqub Davlatov && Habibullo Assoev

Case Study

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

Function Call

Function Template

Compilation Time

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

expr is used to deduce types for T and ParamType

Example

```
template<typename T>
void f(const T& param); // ParamType=const T&
...
int x = 0;
f(x);
```

Example

```
template<typename T>
void f(const T& param); // ParamType=const T&
...
int x = 0;
f(x);
```

Spoiler!!!
T=>int
ParamType=>const int&

Easy right? Hold up your horses

```
template<typename T>
void f(const T& param); // ParamType=const T&
...
int x = 0;
f(x);
```

Just replace T with int???
That easy???

Note !!!

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

The type of T depends both on the **type of** expr and the **form of** ParamType

3 base cases for type deduction

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

- A pointer or reference type
- A universal reference
- Neither a pointer nor a reference

Case 1

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

ParamType is a pointer or reference type

How type deduction will work?

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

Step 1: if expr's type is a reference, ignore the reference part

How type deduction will work?

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

Step 1: if expr's type is a reference, ignore the reference part

Step 2: pattern-match expr's type against ParamType to determine T

Example

```
template<typename T>
void f(T& param);
...
int x = 1;
const int cx = x;
const int& rx = x;

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

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

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

Step 1: if expr's type is a reference, ignore the reference part

Step 2: pattern-match expr's type against ParamType to determine T

Example

```
template<typename T>
void f(const T& param);
...
int x = 1;
const int cx = x;
const int& rx = x;

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

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

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

Step 1: if expr's type is a reference, ignore the reference part

Step 2: pattern-match expr's type against ParamType to determine T

Example

```
template<typename T>
void f(T* param);
...
int x = 1;
const int* px = &x;
const int* const cpx = px;

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

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

f(cpx); // param's type is const int* const
        // T is const int
```

Step 1: if expr's type is a reference, ignore the reference part

Step 2: pattern-match expr's type against ParamType to determine T

Case 2

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

ParamType is a universal reference

How type deduction will work?

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

Step 1: If expr is an **lvalue**, both T and ParamType are deduced to be **lvalue references**

How type deduction will work?

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

Step 1: If expr is an **lvalue**, both T and ParamType are deduced to be **lvalue references**

Step 2: If expr is an **rvalue**, the "normal" (i.e., Case 1) rules apply

Example

```
template<typename T>
void f(T&& param);
...
int x = 1;
const int cx = x;
const int& rx = x;

f(x);    // param's type is int&
         // T is int&
f(cx);   // param's type is const int&
         // T is const int&
f(rx);   // param's type is const int&
         // T is const int&
f(1);    // param's type is int&&
         // T is int
```

Step 1: If expr is an lvalue, both T and ParamType are deduced to be **lvalue references**

Step 2: If expr is an rvalue, the "normal" (i.e., Case 1) rules apply

Case 3

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

ParamType is neither a pointer nor a reference

Case 3

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

param will be a **copy** of whatever is passed in

How type deduction will work?

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

Step 1: if expr's type is a reference, ignore the reference part

How type deduction will work?

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

Step 1: if expr's type is a reference, ignore the reference part

Step 2: if expr is const, ignore that, too

Complete example

```
template<typename T>
void f(T param);
...
int x = 1;
const int cx = x;
const int& rx = x;
```

Step 1: if expr's type is a reference, ignore the reference part
Step 2: if expr is const, ignore that, too

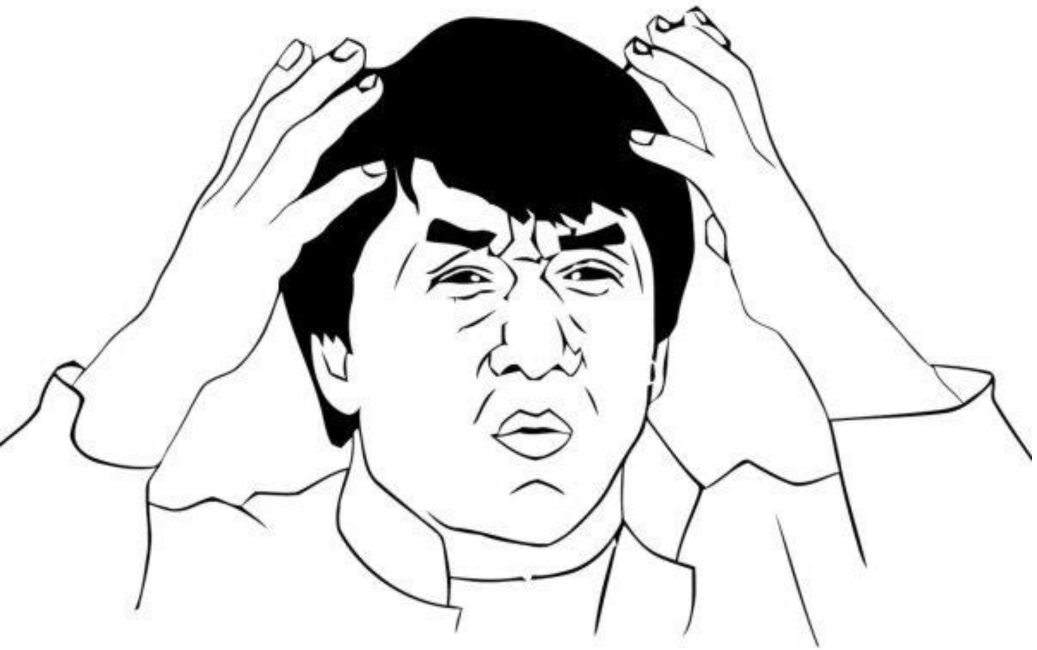
```
f(x);   // T's and param's types are both int

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

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

auto type deduction

When a variable is declared using auto, auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

auto type deduction

When a variable is declared using auto, auto plays the role of T in the template, and the type specifier for the variable acts as ParamType



You after reading this :)
Confusion or Realization?
Maybe both?

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
auto x = 1;


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


func_for_x(1);
```

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
auto x = 1;

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

func_for_x(1);
```

Case 3:

neither a pointer nor a reference

Step 1: if expr's type is a reference, ignore the reference part
Step 2: if expr is const, ignore that, too

ParamType is int
T is int
auto is int
x is int

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
const auto cx = x;


template<typename T>
void func_for_cx(const T param);


func_for_cx(x);
```

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
const auto cx = x;

template<typename T>
void func_for_cx(const T param);

func_for_cx(x);
```

Case 3:

neither a pointer nor a reference

Step 1: if expr's type is a reference, ignore the reference part
Step 2: if expr is const, ignore that, too

ParamType is const int
T is int
auto is int
cx is const int

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
const auto& rx = x;


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


func_for_rx(x);
```

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
const auto& rx = x;


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


func_for_rx(x);
```

Case 1:

a pointer or reference type

Step 1: if expr's type is a reference, ignore the reference part
Step 2: pattern-match expr's type against ParamType to determine T

ParamType is const int&
T is int
auto is int
rx is const int&

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
auto&& ux = x;


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


func_for_ux(x);
```

Let's break it down

auto plays the role of T in the template, and the type specifier for the variable acts as ParamType

```
auto&& ux = x;


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


func_for_ux(x);
```

Case 2:

a universal reference

Step 1: If expr is an lvalue, both T and ParamType are deduced to be **lvalue references**

Step 2: If expr is an rvalue, the "normal" (i.e., Case 1) rules apply

ParamType is int&

T is int&

auto is int&

ux is int&

auto rules == template rules ???

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

```
template<typename T>
void f(T param);
f({ 11, 23, 9 }); // error! can't deduce type for T
```

auto rules == template rules ???

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

template<typename T>
void f(std::initializer_list<T> param);
f({ 11, 23, 9 }); // ok! T is int
```

auto rules == template rules ???

auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.

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

auto rules == template rules ???

auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.

```
std::vector<int> v;
auto resetV = [&v](const auto& newValue) { v = newValue; };
resetV({ 1, 2, 3 });  // error: can't deduce type for { 1, 2, 3 }
```