Template Type Deduction Modern C++

By
SimplyProgram.Hub

Ashish Shelke

Why Type Deduction?

- In C++98, there was only one set of rules for type deduction, specifically for templates.

- C++11 introduced auto and decltype, enhancing the language by:
    - •Reducing redundant type declarations
    - •Propagating type changes across the code automatically But, type deduction can make code harder to read. Let’s explore how it works!

Scenarios in Template Type Deduction

- When deducing types in template functions, C++ follows these three main cases:

```
template<typename T>
void f(paramType param);
f(expr); // Deduce T and ParamType from expr
```

1) ParamType is a **Pointer or reference (non-universal)**
2) ParamType is a **Universal reference**
3) ParamType is a **Pass-by-value (neither pointer nor reference)**

- Each of these cases influences how types are deduced in subtle but important ways.

Case 1 - Pointer or Reference (Non-Universal)

- Type deduction steps as follows:

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

Example 1: Param is 'reference'

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

Code	T's is deduced as	param's type is deduced as
int x = 27; f(x);	int	int&
const int cx = x; f(cx);	const int	const int&
const int& rx = x; f(rx);	const int	const int&

- when programmer passes a const object to reference parameter, they expect that object to be unmodifiable, i.e. parameter to be reference to const. Passing const object to template taking T& as parameter is safe.
- Constness of the object becomes part of the type deduced for T.

Case 1 - Pointer or Reference (Non-Universal)

Example 2: Param is 'reference to const’

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

Code	T’s is deduced as	param’s type is deduced as
int x = 27; f(x);	Int	const int&
const int cx = x; f(cx);	Int	const int&
const int& rx = x; f(rx);	Int	const int&

Example 3: Param is pointer or pointer to const instead reference’

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

Code	T’s is deduced as	param’s type is deduced as
int x = 27; f(&x);	int	Int*
const int* px = &x; f(px);	const int	const int*

Case 2 - Universal References

- Universal references (T&&) can handle both l-values and r-values.
- They behave differently based on the type of expression passed:
  - For l-values, both T and ParamType are deduced to be l-value references.
  - For r-values, deduction follows the case-1 rules.

Example: Param is 'universal reference'

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

Code	Reference type	T's is deduced as	param's type is deduced as
int x = 27; f(x);	x is l-value	Int&	int&
const int cx = x; f(cx);	cx is l-value	const int&	const int&
const int& rx = x; f(rx);	rx is l-value	const int&	const int&
f(27);	27 is r-value	int	Int&&

Case 3 - Pass-by-Value

- Type deduction steps as follows:

1. If expr's is a reference, ignore the reference part
2. If after ignoring reference-ness, expr is const, then ignore that too. If its volatile also ignore that.

Example 1: Param is ‘pass by value’

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

Code	T’s is deduced as	param’s type is deduced as
int x = 27; f(x);	int	int
const int cx = x; f(cx);	int	int
const int& rx = x; f(rx);	int	int

- Even though cx and rx represents const values, param isn't const. That makes sense. Param is object that is completely independent of cx and rx as its copy of cx and rx. Function will modify copied objects.
- That's why expr's constness and volatileness is ignored

Case 3 - Pass-by-Value

Example 2: Param is ‘const pointer to const object and expr is passed by value’

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

Code	T’s is deduced as	param’s type is deduced as

const char* const ptr = "Fun with Pointers";
f(ptr);
	char*	const char*

- As per type deductions steps, pointer to const object is passed to parameter as pass by value. So const ptr is able to point another object so constness of pointer is ignored here.
- This should be avoided