Function Template Type Deduction

Contents

Version

Version	Author	Date	Changes
0.1	Jin Feng	2017.01.15	Initial Draft
0.9	Jin Feng	2020.01.30	Rearrange the structure and format
1.0	Jin Feng	2020.01.31	Add sumarization and appendix sections, sort out structure

Overview

We have function template and class template. Note that class template parameters are never deduced. The reason is that the flexibility provided by several constructors for a class would make such deduction impossible in many cases and obscure in many more. So we consider function template type deduction in this article.

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

A call can look like this:

```
ExprType expr;
f(expr); // call f with some expression
```

So the problem is how to deduce T from ParamType and ExprType. That is to say:

> T = F(ExprType, ParamType)

The ExprType could include ***literal<1>, plain variable<int>, lvalue refernce<int&>, rvaule reference<std::move(x)>, pinter<int*>, array<int[10]>, and function<void(int)>*** with/without qualifiers. To understand function types well please refer to article "*C Function Pointer*".

Cases

Case 1

Case 1: ParamType is a Reference or Pointer, but not a Universal Reference Type deduction work like this:

> *T is deduced to the one which is missed in ParamType compared with expr's type.*

Example 1: T&

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

int x = 27;
const int = cx =x;
const int& rx =x;

const char name[] = "J. P. Briggs";
const char * ptrToName = name;

void someFunc(int, double);

f(x);
f(cx);
f(rx);
f(name);
f(someFunc);
```

expr	ExprType	ParamType	T deduced	ParamType deduced	explanation
x	int	T&	int	int &	missing int
cx	const int	T&	const int	const int &	missing const int
rx	const int&	T&	const int	const int &	missing const int
name	const char [13]	T&	const char [13]	const char (&) [13]	missing const char [13]
someFunc	void (int, double)	T&	void (int, double)	void (&)(int, double)	missing void (int, double)

Example 2: T const &

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

int x = 27;
const int = cx =x;
cont int& rx =x;

f(x);
f(cx);
f(rx);
```

expr	expr's type	ParamType	T deduced	ParamType Deduced	explanation
x	int	const T&	int	const int&	missing int
cx	const int	const T&	int	const int&	missing int
rx	const int&	const T&	int	const int&	missing int

Example 3: T*

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

int x = 27;
const int *px = &x;

f(&x);
f(px);
```

expr	expr's type	ParamType	T deduced	ParamType Deduced	explanation
x	int *	T*	int	int*	missing int
px	const int *	T*	const int	const int *	missing const int

Case 2

Case 2: ParamType is a Universal Reference(forwarding reference)

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

Please note T&& is a universal declaration, it has nothing to do with rvalue references. Type deduction work like this:

- If expr is lvalue, append "&" to expr's type, no need for rvalue, we call it expr's extended type
- T is deduced to the one which is missed in ParamType compared with expr's extended type

The following identies shall be used while deducing T and ParamType:

> We take & as 1 and && as 0 for easy remebering, so we will have
> & != && // 1 != 0
> & & != && // 1 | 1 != 0
> & & = & // 1 | 1 = 1
> && & = & // 0 | 1 = 1

Please note that & is different from &&

For example:

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

int x = 27;
const int cx = x;
const int& rx = x;

f(x);
f(cx);
f(rx);
f(27);
```

expr	expr's type	expr's extended type	ParamType	T deduced	ParamType deduced	Note
x	int	int &	T &&	int &	int & && = int &	-
cx	const int	const int &	T &&	const int &	const int & && = const int &	-
rx	const int &	const int & & = const int &	T &&	const int &	const int & && = const int &	-
27	int	int	T&&	int	int &&	-

Case 3

Case 3: ParamType is Neither a Pointer nor a Reference

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

That means that param will be a copy of whatever is passed in - a completely new object. We ignore ***top level*** reference, modifiers(const/volatile) of expr's type, just because expr can't be modified doesn't mean that a copy of it can't be.

Example 1:

```
int x = 27;
const int cx = x;
const int& rx = x;
const char* const ptr = "Fun with pointers";
void someFunc(int, double);

f(x);
f(cx);
f(rx);
f(ptr);
f(someFunc);
```

expr	expr's type	ref&modfies ignored	ParamType	T deduced	Note
x	int	int	T	int	-

expr	expr's type	ref&modfies ignored	ParamType	T deduced	Note
cx	const int	int	T	int	-
rx	const int &	int	T	int	-
ptr	const char * const	const char *	T	const char *	ignore top level const: ***const char * ~~const~~***
someFunc	void (*)(int, double)	void (*)(int, double)	T	void (*)(int, double)	function type decay into function pointers, same as array(int array[10])

Summary

For completeness, let us give a sumarization for all ExprTypes which could be ***literal<1>, plain variable<int>, lvalue refernce<int&>, rvaule reference<std::move(x)>, pinter<int*>, array<int[10]>, and function<void(int)>***

Table: Function Template Type Deduction

ExprType	expr	void f(T param)		void f(T* param)		void f(T& param)		void f(T&& param)	
		T	ParamType	T	ParamType	T	ParamType	T	ParamType
literal	1	int	int	int	int*	-	-	int	int&&
plain	int x	int	int	-	-	int	int&	int&	int&
lvalue ref	int &rx=x	int	int	-	-	int	int&	int&	int&
rvalue ref	f(std::move(x))	int	int	-	-	-	-	int	int&&
pointer	int *px=&x	int*	int*	int	int*	int*	int*&	int* &	int* &
array	int ax[1]	int*	int*	int	int*	int [1]	int (&) [1]	int (&) [1]	int (&) [1]
function	void func(){}	void(*)()	void(*)()	void(*)()	void()	void ()	void (&)()	void (&)()	void(&()

Appendix

- This article's source code: https://gitee.com/emmix/languages/blob/master/cpp/deducingType/functionTemplateTypeDeduction.md
- The C++ source used in this article: https://gitee.com/emmix/languages/blob/master/cpp/deducingType/deducingType.cpp
- The whole container address: https://gitee.com/emmix/languages/tree/master/cpp/deducingType

References

1. Effective Modern C++, Chapter 1, Item 1
2. The C++ Programming Language, Forth Edition, Chapter 23
3. C Function Pointer