Ref 1 # C++ for C Programmers (Coursera)

Ref 2 # Effective Modern C++ (Scott Meyers)

When you are in a template and a parameter has exactly type T&& for some deduced type T, then what you might get when instantiating the template is not an rvalue reference. Indeed, , the parameter of function can bind to both lvalues and rvalues. On a side note, auto&&works similarly.

> When an array is passed as an argument to function which accepts value by reference, then the parameter in function call is not a pointer rather an array – int (&)[array\_size]

// 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> // see info

constexpr std::size\_t arraySize(T (&)[N]) noexcept // below on

{ // constexpr

return N; // and

} // noexcept

• 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.

• When deducing types for by-value parameters, const and/or volatile argu-

ments 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.

**auto keyword**

array and function names decay into pointers for non-reference type specifiers.

const char name[] = “Gaurav”;

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

C++11 allows a variable declaration in 4 forms:

auto a = 23;

auto a(23);

auto a{23};

auto a = {23};

The first 2 statements declares an integer while the last 2 declares an initializer-list of 1 element as of type: std::initializer\_list<int>

Hence, when auto variable encounters braces, it expects an initializer-list of homogeneous elements. The following statement throws error:

auto x5 = { 1, 2, 3.0 }; // can't deduce T for std::initializer\_list<T> (data doesn’t resolve to a single type)

If similar initializer-list is passed to function template, deduction fails.

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

In C++14, we can omit trailing return type for both functions and lambdas while in C++11 it was possible only for lambdas.

auto, when used as a return type, strip off reference. So, when a function which returns a reference with return type deduction left to auto will fail at the following:

deque<int> q;

auto try(deque<int>& q, int i);

try(q, 12) = 23; // r-value returned, the assignment to which is not allowed

working declaration:

decltype(auto) try(q, 12);

decltype

Given name of expression, decltype gives us the type of parameter passed.

decltype can be used in trailing return type, recently introduced in C++11.

For example: This code gives error because a and b are used before their type is declared.

template<class T>

decltype(a\*b) mul(T a, T b){

return a\*b;

}

while this works fine;

template<class T>

auto mul(T a, T b) -> decltype(a\*b){

return a\*b;

}

**l-values, r-values and move semantics**

An lvalue is an expression e that may appear on the left or on the right hand side of an assignment, whereas an rvalue is an expression that can only appear on the right hand side of an assignment.

An lvalue is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An rvalue is an expression that is not an lvalue.

the return value of a function is an l-value if and only if it is a reference

This is because we have to support chaining, such as

a = b = c;

It means that the assignment operator will have to return a reference.

We want assignment operation to work like this:

// [...]

// swap m\_pResource and rhs.m\_pResource

// [...]

These are called move semantics.

RValue references

If X is any type, then X&& is called an rvalue reference to X. For better distinction, the ordinary reference X& is now also called an lvalue reference.

X x;

X foobar();

foo(x); // argument is lvalue: calls foo(X&)

foo(foobar()); // argument is rvalue: calls foo(X&&)

Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: if it has a name, then it is an lvalue. Otherwise, it is an rvalue.

Here is an example of something that is declared as an rvalue reference and does not have a name, and is therefore an rvalue:

X&& goo();

X x = goo(); // calls X(X&& rhs) because the thing on

// the right hand side has no name

declared asSmart Pointers

Four types

* std::auto\_ptr (deprecated C++11 onwards because it didn’t support move semantics)
* std::unique\_ptr

std::unique\_ptr<int> p1(**new** int(5));

std::unique\_ptr<int> p2 = p1; *//Compile error.*

std::unique\_ptr<int> p3 = std::move(p1); *//Transfers ownership. p3 now owns the memory and p1 is rendered invalid.*

p3.reset(); *//Deletes the memory.*

p1.reset(); *//Does nothing.*

* std::shared\_ptr

std::shared\_ptr<int> p1(**new** int(5));

std::shared\_ptr<int> p2 = p1; *//Both now own the memory.*

p1.reset(); *//Memory still exists, due to p2.*

p2.reset(); *//Deletes the memory, since no one else owns the memory.*

Problem with shared\_ptr

void main( )

{

int\* p = new int;

shared\_ptr<int> sptr1( p);

shared\_ptr<int> sptr2( p );

}

The program will crash

* std::weak\_ptr

Basically used to check the validity of smart pointer if its deleted. shared ownership can be retrieved using weak\_point.lock() function call. It helps resolving cyclic references.

Suppose class A has a shared pointer that points to class B which in turns has another pointer that points to A. The following code will create a cyclic reference and results in memory leak when control goes out of this scope

shared\_ptr<B> sptrB( new B );

shared\_ptr<A> sptrA( new A );

sptrB->m\_sptrA = sptrA;

sptrA->m\_sptrB = sptrB;

which can only be resolved if shared\_ptr is replaced by weak pointer since that will not increase the reference count.

std::shared\_ptr<int> p1(**new** int(5));

std::weak\_ptr<int> wp1 = p1; *//p1 owns the memory.*

{

std::shared\_ptr<int> p2 = wp1.lock(); *//Now p1 and p2 own the memory.*

**if**(p2) *// As p2 is initialized from a weak pointer, you have to check if the memory still exists!*

{

*//Do something with p2*

}

} *//p2 is destroyed. Memory is owned by p1.*

p1.reset(); *//Memory is deleted.*

std::shared\_ptr<int> p3 = wp1.lock(); *//Memory is gone, so we get an empty shared\_ptr.*

**if**(p3)

{

*//Will not execute this.*

}

Sample smart\_ptr class

// A generic smart pointer class

template <class T>

class SmartPtr

{

   T \*ptr;  // Actual pointer

public:

   // Constructor

   explicit SmartPtr(T \*p = NULL) { ptr = p; }

   // Destructor

   ~SmartPtr() { delete(ptr); }

   // overloading dereferencing operator

   T & operator \* () { return \*ptr; }

   // overloading arrow operator so that members of T can be accessed

   // like a pointer (useful if T represents a class or struct or

   // union type)

   T \* operator -> () { return ptr; }

};