C++11中的右值引用：T&&，但是通过T&&定义的不一定都是右值

C++中的右值没有地址，不能修改。在C++11中通过右值引用的临时对象

int c = 0;

// int&& rr1 = c; // error，右值引用不能绑定左值

int&& rr1 = std::*move*(c); // ok，move()将左值c强制转换为右值

rr1 = 2;

*cout* << c << *endl*; // 2

例：程序stl\_test5

int a = 5;

const int b = 6;

int c = 7;

// 非常量右值引用可以绑定到非常量右值

int&& r1 = 4; // 非常量右值

r1 = 5; // 可以这样写代码，但没有意义

//int&& r2 = a; // error，右值引用不能绑定到左值

int&& r3 = a + c; // 临时对象，非常量右值

// 常量右值引用可以绑定到非常量右值和常量右值

// 常量右值引用本身没有现实意义

const int&& r4 = 5;

const int&& r5 = a + b; // 常量右值

//const int&& r6 = a; // error，不能绑定到左值

// 注意：绑定到右值的右值引用本身是一个左值

// 因为它有名字

//int&& r7 = r1; // error，不能绑定到左值

int&& r8 = *move*(r1); // ok，通过move将左值变成右值

右值引用的第一个重要作用是移动语义，解决C++在拷贝临时对象时不必要的资源拷贝。

使用std::move后的对象最好不要再使用，因为其资源已经被搬移走了。

例：程序stl\_test6

class P

{

public:

P()

{

*cout* << "构造函数" << *endl*;

}

P(const P& p)

{

*cout* << "拷贝构造函数" << *endl*;

}

~P()

{

*cout* << "析构函数" << *endl*;

}

};

int *main*(int argc, char\* argv[])

{

*vector*<P> vec\_p;

P p1;

// vector容器在插入p1时

// 会首先调用拷贝构造函数

// 构造一个临时对象插入容器中

vec\_p.*push\_back*(p1);

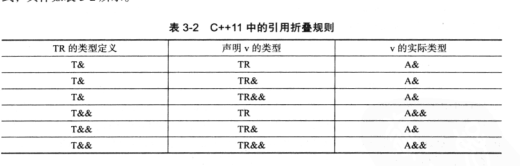
return 0;

}

先了解C++11中的引用叠加规则。

普通函数的形参如果定义为右值引用的形式，则不能接受左值。但函数模板例外，所以就有了引用叠加规则。对于模板类型，还有一个模板参数类型规则。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 调用前形参声明的类型 | 调用时传入的实参类型 | 调用后形参的实际类型 | 模板参数的实际类型 |
| 1 | T& | A& | A& | A |
| 2 | T& | A&& | A& | A |
| 3 | T&& | A& | A& | A& |
| 4 | T&& | A&& | A&& | A |



规则1验证：形参formal\_param声明为T&，实参actual\_param2定义为A&

例：程序stl\_test7

template<typename T>

void FuncRule1(T& formal\_param)

{

T formal\_param1 = 0;

*cout* << "验证规则1" << *endl*;

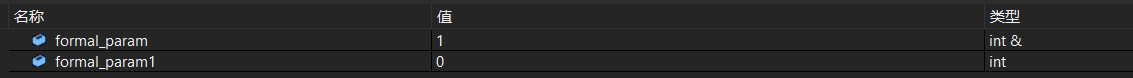
}

int actual\_param1 = 1;

int& actual\_param2 = actual\_param1;

FuncRule1 (actual\_param2);

断点调试的截图：



从图中可以看出，形参formal\_param的实际类型为int&，即A&，而模板类型T的实际类型从formal\_param1可以看出为int，即A。

规则2验证：形参formal\_param声明为T&，实参actual\_param3定义为A&&

bool IsRightReference(int&& r)

{

*cout* << "右值" << *endl*;

return true;

}

bool IsRightReference(int& l)

{

*cout* << "左值" << *endl*;

return false;

}

// 形参需要加const

// 接收的实参为右值

template<typename T>

void FuncRule2(const T& formal\_param)

{

T formal\_param1 = 0;

*cout* << "验证规则2" << *endl*;

}

int&& actual\_param3 = *move*(actual\_param1);

IsRightReference(actual\_param3); // 左值

IsRightReference(*forward*<int>(actual\_param3)); // 右值

FuncRule2(*forward*<int>(actual\_param3));

断点调试截图：



从图中可以看出，形参formal\_param的实际类型为const int&，即A&，而模板类型T的实际类型从formal\_param1可以看出为int，即A。

规则3验证：形参formal\_param声明为T&&，实参actual\_param2定义为A&

template<typename T>

void FuncRule3(T&& formal\_param)

{

int n = 0;

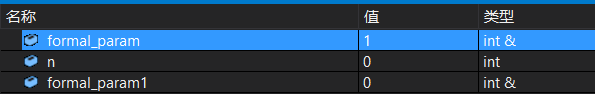
T formal\_param1 = n;

*cout* << "验证规则3" << *endl*;

}

FuncRule3(actual\_param2);

断点调试截图：



从图中可以看出，形参formal\_param的实际类型为int&，即A&，而模板类型T的实际类型从formal\_param1可以看出为int&，即A&。

规则4验证：形参formal\_param声明为T&&，实参actual\_param3定义为A&&。

template<typename T>

void FuncRule4(T&& formal\_param)

{

int n = 0;

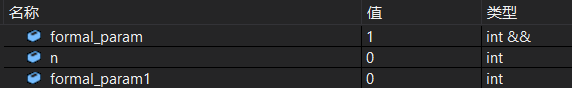
T formal\_param1 = n;

*cout* << "验证规则4" << *endl*;

}

FuncRule4(*forward*<int>(actual\_param3));

断点调试截图：



从图中可以看出，形参formal\_param的实际类型为int&&，即A&&，而模板类型T的实际类型从formal\_param1可以看出为int，即A。

右值引用的强制转换：

使用std::forward<T>可以将左值强制转换为右值。

返回右值引用。

例：程序stl\_test8

class P

{

public:

P() : length\_(10), ptr\_(new int[length\_])

{

*cout* << "default constructor" << *endl*;

for (int i = 0; i < length\_; ++i)

{

ptr\_[i] = i;

}

}

P(const P& p) : length\_(p.length\_)

{

*cout* << "copy constructor" << *endl*;

ptr\_ = new int[length\_];

for (int i = 0; i < length\_; ++i)

{

ptr\_[i] = p.ptr\_[i];

}

}

P(P&& p) : length\_(p.length\_), ptr\_(p.ptr\_)

{

*cout* << "P&& constructor" << *endl*;

p.ptr\_ = nullptr;

}

~P()

{

*cout* << "~P()" << *endl*;

if (ptr\_)

{

delete[] ptr\_;

}

ptr\_ = nullptr;

}

private:

int length\_;

int\* ptr\_;

};

P& Func1()

{

P p;

P& rp = p;

return rp;

}

P&& Func2()

{

P p;

return *move*(p);

}

int *main*(int argc, char\* argv[])

{

// error,Func1()返回引用

// 但p是一个局部变量，返回左值引用rp所引用的对象

// 会马上析构掉，p1的拷贝构造函数传入的是

// 一个已析构的对象

//P p1(Func1());

// error，Func2()将局部变量p从左值变成右值引用

// 右值引用也是引用，会和上述的左值引用出现同样的问题

//P p2(Func2());

return 0;

}

完美转发：

函数模板在向其他函数传递参数时保留参数的const属性、左右值属性，即函数模板在向其他函数传递自身形参时，如果实参是左值，就转发为左值，如果实参为右值，就转发为右值，实参为const值，转发也为const值。

例：程序stl\_test9

void Fun(int& x)

{

*cout* << "lvalue ref" << *endl*;

}

void Fun(int&& x)

{

*cout* << "rvalue ref" << *endl*;

}

void Fun(const int& x)

{

*cout* << "const lvalue ref" << *endl*;

}

void Fun(const int&& x)

{

*cout* << "const rvalue ref" << *endl*;

}

// 完美转发函数模板

template <typename T>

void PerfectForward(T&& t)

{

Fun(*forward*<T>(t));

}

int *main*(int argc, char\* argv[])

{

PerfectForward(10);

int a = 5;

PerfectForward(a);

PerfectForward(*move*(a));

const int b = 8;

PerfectForward(b);

PerfectForward(*move*(b));

return 0;

}

输出结果为：

rvalue ref

lvalue ref

rvalue ref

const lvalue ref

const rvalue ref