第13课 右值引用

Posted on 2019-08-06 12:22 浅墨浓香 阅读(622) 评论(2) 编辑 收藏 举报

一. 左值和右值

（一）概述

1. 左值是一般指表达式结束后依然存在的持久化对象。右值指表达式结束时就不再存在的临时对象。**便捷的判断方法：能对表达式取地址、有名字的对象为左值。反之，不能取地址、匿名的对象为右值。**

2. C++ 表达式（运算符带上其操作数、字面量、变量名等）有**两种独立的属性：类型和值类别 (value category)。类型指变量声明时的类型。**而**值类别是表达式结果的类型，它必属于左值、纯右值或将亡值三者之一。**如int&& x;其中x的类型为右值引用，但作为表达式**使用时**其值类别为左值（因为有名字，可以取址）

（二）右值的分类

1. 纯右值(prvalue)：用于识别临时变量和一些不与对象关联的值。函数返回值为非引用类型、表达式临时值(如1+3)、lambda表达式等。

2. 将亡值(xvalue)：是**与右值引用相关的表达式，**通常指将要被移动的对象。如，函数返回类型为T&&、std::move的返回值、转换为T&&的类型转换函数的返回值（**注意，这些都是与右值引用相关的表达式**）或临时对象。

（三）表达式值类别的典型例子

	左值表达式(lvalue)	右值表达式(rvalue)	
		纯右值表达式(prvalue)	将亡值表达式(xvalue)
名字	由变量、函数或数据成员等**名字构成的表达式**仍为左值表达式(注意，不论其类型)。**如具名的右值引用为左值表达式**。	匿名对象。如**匿名的右值引用**。	
函数调用或者重载运算符	①**返回类型是左值引用的。** ②返回类型是到函数的右值引用。	返回类型是**非引用类型**。	返回类型为**对象的右值引用**，例如 std::move(x)
类型转换表达式	①**转换为左值引用类型，如static_cast<int&>(x)** ②转换为函数的右值引用类型的转型表达式，如static_cast<void (&&)(int)>(x)也是左值表达式	**转换为非引用类型**的转型表达式如static_cast<double>(x)、(int)42等。	转换为**对象的右值引用类型的转型表达式**，例如static_cast<char&&>(x)
对象成员表达式(a.m)	**①m为静态成员函数** **②a 为左值且 m 为非引用类型的非静态数据成员**	**①m为普通成员函数（只能用于函数调用，如a.m(10)，不能用于初始化引用或作为函数实参等其它用途）** **②a 为prvalue且 m 为非引用类型的非静态数据成员**	a 是xvalue且 m 是非引用类型的非静态数据成员
字面量	**字符串字面量**（如”hello world!”	除字符串字面量之外的字面量。如42、true、nullptr	无
其它	①内置赋值表达式（如a = b、a+= b等； ②**间接寻址表达式（如*p）**	①内置算术、逻辑和比较表达式 ②内置**取地址表达式**（如&a） ③**lambda表达式**	**临时对象的表达式**

二、左值引用和右值引用

（一）概述

1. 左值引用和右值引用都属于引用类型。无论是声明一个左值引用还是右值引用都必须立即进行初始化。

2. **左值引用都是左值。**但**具名的右值引用是左值，而匿名的右值引用是右值。**

（二）可绑定的值类型(设T是个具体类型)

1. **左值引用（T&）：只能绑定到左值**（非常量左值）

2. **右值引用（T&&）：只能绑定到右值**（非常量右值）

3. **常量左值引用（const T&）**：常量左值引用是个**“万能”的引用类型。它既可以绑定到左值也可以绑定到右值。**它像右值引用一样可以延长右值的生命期。不过相比于右值引用所引用的右值，常量左值引用的右值在它的“余生”中只能是只读的。

4. 常量右值引用(const T&&)：可绑定到右值或常量右值。由于移动语义需要右值可以被修改，因此常量右值引用没有实际用处。如果需要引用右值且让其不可更改，则常量左值引用就足够了。

【编程实验】左值引用和右值引用

```
#include <iostream>
#include <vector>
using namespace std;

class Widget
{
public:
    int x;
    int& rx = x;

    int arr[10];

    static int staticfunc(int x)
    {
        cout << "static int Widget::staticfunc(int x): " <<x << endl;
        return x;
    }

    int commonfunc(int x)
    {
        cout << "int Widget::commonfunc(int): " << x << endl;
        return x;
    }
};

Widget makeWidgetR()
{
    return Widget();
}

Widget& makeWidgetL()
{
    static Widget w;
    return w;   //ok， Widget&是个引用类型，要注意不能返回局部对象。
}

Widget&& makeWidgetX()
{
    static Widget w;
    return std::move(w);  //ok。但要注意，Widget&&是个引用类型不能返回局部对象。
}

//返回到函数的引用类型
using RetFunc = int(int);
int demoImpl(int i)
{
    cout << "int demo(int): " << i << endl;
    return i;
}

RetFunc&& RetFuncDemo()
{
    return demoImpl;
}

int main()
{
    //1. 常见的左/右值表达式分析

    //1.1 函数形参为左值
    //int test(int&& x){return x;} //形参x为左值（具名变量），尽管其为右值引用类型。

    int i = 0;
    //1.2 前置/后置自增、自减表达式
    int&& ri = i++;  //i++为右值，表达式返回的是i的拷贝，匿名对象是个右值
    int& li = ++i;   //++i返回i本身，是个具名对象，为左值。

    int& r2 = ri;    //虽然ri的类型是int&&，但ri是个具名变量为左值。
    //int&& r3 = ri; //error,ri是个左值
    r2 = 5;
    cout << "ri = " << ri << ", i = "<< i << endl;  //5, 2

    //1.3 解引用和取地址运算表达式
    int* p = &i;
    int& lp = *p;   //解引用：*p为左值，因为可以对*p取址址&(*p)。或*p = 5;
    int*&& rp = &i; //取地址：&i是个内存地址，是个右值。可以用来初始化右值引用
    *rp = 10;
    cout  <<"i = "<< i << endl; //i = 10，i的值通过rp引用修改。

    //1.4 字面量
    const char(&hw)[13] = "hello world!"; //字符串字面量是左值，可以用于初始化左值引用
    cout << "const char(&hw)[13] = " << hw << endl;

    int&& ten = 10;  //10为纯右值

    //1.5 赋值表达式 和 算术表达式、比较表达式、逻辑表达式
    int& a = (i += 2);  //i +=2为赋值表达式，结果为左值。类似的，还有 a = b、a %= b
    int&& b = i + 2;    //i+2为算术表达式，结果为右值。类似的还有a + b、a % b、a & b、a << b
    int&& c = (a > b);  //比较表达式结果为右值。类似的还有：a < b、a == b、a >= b
    int&& d = (a && b); //逻辑表达式结果为右值。类似的还有：a && b、a || b、!a

    //1.6 lambda表达式
    //auto& lam = [](int x, int y) {return x + y; };//lambda表达式为纯右值，不能绑定到左值

    //2.下标表达式
    int arr[10];
    int& ra1 = arr[2]; //[]下标表达式返回左值引用，仍是左值
    vector<int> vec{ 1,2,3,4 };
    int& rv = vec[2];  //operator[]返回左值引用，是个左值表达式。

    //3.对象访问表达式
    Widget w1;
    int& rx1 = w1.x;          //w1为左值，所以w1.x为左值
    int&& rx2 = Widget().x;   //Widget()是个临时对象（右值）。因此，Widget().x为右值。
    //int& rx3 = Widget().x; //error，理由同上。

    using WidgetStaticFunc = int(int);
    WidgetStaticFunc& wsf = w1.staticfunc; //静态成员函数，是个左值。
    wsf(10);
    w1.commonfunc(2);   //w1.commomfunc是个纯右值，只能用于函数调用，不能做其它用途。

    //4. 类型转换表达式
    int&& i1 = std::move(i);              //std::move()返回值为右值引用类型，是个右值。
    int&& i2 = static_cast<int&&>(i);     //转换为右值引用类型，表达式结果是个右值
    double&& d1 = static_cast<double>(i); //转换为右值类型，结果是个右值
    int& i3 = static_cast<int&>(i);       //转换为左值引用类型，表达式结果为左值

    //5.函数返回型类型
    Widget&  w2 = makeWidgetL();   //返回左值引用类型，为左值表达式
    Widget&& w3 = makeWidgetR();   //返回非引用类型，为右值。w3为右值引用，可以引用右值（makeWidgetR的返回值）
    Widget&& w4 = makeWidgetX();   //返回右值引用类型，为右值表达式
    RetFunc& rf1 = RetFuncDemo();  //RetFuncDemo返回一个到函数的右值引用，是个左值表达式(C++11的标准行为）。
    RetFunc&& rf2 = RetFuncDemo(); //RetFuncDemo仍可以用来初始化右值引用！
    rf1(5);

    return 0;
}
/*输出结果
ri = 5, i = 2
i = 10
const char(&hw)[13] = hello world!
static int Widget::staticfunc(int x): 10
int Widget::commonfunc(int): 2
int demo(int): 5
*/
```

三、万能引用（universal reference）

（一）T&&的含义

1. 当T是一个**具体的类型**时，**T&&表示右值引用**，只能绑定到右值。

2. 当**涉及T类型推导时，T&&为万能引用**。若用右值初始化万能引用，则T&&为右值引用。若用左值初始化万能引用，则T&&为左值引用。但**不管哪种情况，T&&都是一种引用类型。**

（二）万能引用

1. **T&&是万能引用的两个条件**：

   （1）必须涉及类型推导；

   （2）声明的形式也必须正好形如“T&&”。并且该形式被限定死了，任何对其修饰都将剥夺T&&成为万能引用的资格。

2. 万能引用使用的场景

   （1）函数模板形参

   （2）auto&&

【编程实验】万能引用

```
#include <iostream>
#include <vector>
using namespace std;

class Widget {};

void func1(Widget&& param) {};  //param为右值引用类型(不涉及类型推导)

template<typename T>
void func2(T&& param){} //param为万能引用（涉及类型推导）


template<typename T>
void func3(std::vector<T>&& param) {} //param为右值引用，因为形式不是正好T&&
                                      //param的类型已确定为vector类型，而推导的是其元素的类型，
                                      //而不是param本身的类型。

template<typename T>
void func4(const T&& param){}  //param是个右值引用，因为被const修饰，其类型为const T&&，而不符”正好是T&&”的要求

template<class T>
class MyVector
{
public:
    void push_back(T&& x){} //x为右值引用。因为当定义一个MyVector对象后，T已确定。当调用该函数时T的类型不用再推导！
                            //如MyVector<Widget> v; v.push_back(...);时T已经是确定的Widget类型，无须再推导。
    template<class...Args>
    void emplace_back(Args&& ... args) {}; //args为万能引用，因为Args独立于T的类型，当调用该函数时，需推导Args的类型。
};


int main()
{
    //1. 模板函数形参（T&&）
    Widget w;
    func2(w); //func2(T&& param)，param为Widget&（左值引用）
    func2(std::move(w)); //param为Widget&&，是个右值引用。

    //2. auto&&
    int x = 0;
    Widget&& var1 = Widget();  //var1为右值引用（不涉及类型推导）
    auto&& var2 = var1;        //万能引用，auto&&被推导为Widget&（左值引用）
    auto&& var3 = x;           //万能引用，被推导为int&;(左值引用）

    //3. 计算任意函数的执行时间：auto&&用于lambda表达式形参（C++14）
    auto timefunc = [](auto && func, auto && ... params)
    {
        //计时器启动

        //调用func(param...)函数
        std::forward<decltype(func)>(func)(           //根据func的左右值特性来调用相应的重载&或&&版本的成员函数
            std::forward<decltype(params)>(params)... //保持参数的左/右值特性
            );

        //计时器停止并记录流逝的时间
    };

    timefunc(func1, std::move(w)); //计算func1函数的执行时间

    return 0;
}
```