第7章 按值传递或按引用传递：7.3 使用std::ref()和std::cref()

Posted on 2020-05-02 17:50 浅墨浓香 阅读(371) 评论(0) 编辑 收藏 举报

7.3 Using std::ref() and std::cref()

7.3 使用std::ref()和std::cref()

Since C++11, you can let the caller decide, for a function template argument, whether to pass it by value or by reference. When a template is declared to take arguments by value, the caller can use std::cref() and std::ref(), declared in header file <functional>, to pass the argument by reference. For example:

从C++11开始，对于函数模板参数，你可以让调用者自己决定是按值还是按引用来传递。当模板被声明按值传递时，调用者可以使用std::cref()和std::ref()（声明在<functional>头文件中）将参数按引用传递给函数模板。例如：

```
template<typename T>
void printT (T arg) {
    …
}

std::string s = "hello";
printT(s); //按值传递s
printT(std::cref(s)); // 模拟按引用传递s
```

However, note that std::cref() does not change the handling of parameters in templates. Instead, it uses a trick: It wraps the passed argument s by an object that acts like a reference. In fact, it creates an object of type std::reference_wrapper<> referring to the original argument and passes this object by value. The wrapper more or less supports only one operation: an implicit type conversion back to the original type, yielding the original object. So, whenever you have a valid operator for the passed object, you can use the reference wrapper instead. For example:

但是，请注意std::cref()并没有改变模板内部处理参数的方式。相反，它使用了一个技巧：它用一个“可以被看作是引用”的对象，将原始参数包装了起来。实际上，它创建了一个 std::reference_wrapper<> 对象，该对象会引用原始参数，并将自己以传值的形式，传递给了函数模板。这种包装器对象，或多或少地支持一个操作：将自己隐式转换为一个原始参数类型，并返回原始参数对象（译注：也可以使用成员函数get() 来获得原始参数的左值引用类型。如果原始对象是可调用的，更可以使用 operator()来调用它）。也就是说，如果原始对象支持某种操作，那么这个包装器对象也就能执行这个操作。例如：

```
#include <functional> // for std::cref()
#include <string>
#include <iostream>

void printString(std::string const& s)
{
    std::cout << s << '\n';
}

template<typename T>
void printT (T arg)
{
    printString(arg); // 可能将arg转换成std::string
}

int main()
{
    std::string s = "hello";
    printT(s); // 打印s （按值传递）
    printT(std::cref(s)); // 打印s（就像按引用传递）
}
```

The last call passes by value an object of type std::reference_wrapper<string const> to the parameter arg, which then passes and therefore converts it back to its underlying type std::string.

代码中的最后一次调用 printT()，是以传值的形式，将一个 std::reference_wrapper<string const> 对象传递给参数 arg。这个包装器对象会被进一步传递，最后在调用 printString() 函数时，被转换为底层的std::string类型。

Note that the compiler has to know that an implicit conversion back to the original type is necessary. For this reason, std::ref() and std::cref() usually work fine only if you pass objects through generic code. For example, directly trying to output the passed object of the generic type T will fail because there is no output operator defined for std::reference_wrapper<>:

注意，编译器需要被告知，这个隐式类型转换是“有必要的”。因此，通常只有在模板代码中传递对象时，std::ref / std::cref 的隐式类型转换才会运作正常。例如，如果尝试在printT()函数中直接输出arg对象(泛型T类型)则会失败（译注：std::cout重载中并没有string&形参版本，编译器就不会得到任何需要进行隐式转换的提示，类型T就不会自动转换回原始参数类型），因为std::reference_wrapper<>类并没有重载operator<<运算符

```
template<typename T>
void printV (T arg) {
    std::cout << arg << '\n';
}

…

std::string s = "hello";
printV(s); //OK
printV(std::cref(s)); // ERROR: std::reference_wrapper<>类没有重载operator <<
```

Also, the following fails because you can’t compare a reference wrapper with a char const* or std::string:

同样，下面代码也会报错，因为无法将std::reference_wrapper<>对象与char const*或者std::string进行比较：

```
template<typename T1, typename T2>
bool isless(T1 arg1, T2 arg2)
{
    return arg1 < arg2;
}

…

std::string s = "hello";
if (isless(std::cref(s), "world")) … //ERROR
if (isless(std::cref(s), std::string("world"))) … //ERROR
```

It also doesn’t help to give arg1 and arg2 a common type T:

即使arg1和arg2的具有相同的模板参数类型T，也无济于事：

```
template<typename T>
bool isless(T arg1, T arg2)
{
    return arg1 < arg2;
}
```

because then the compiler gets conflicting types when trying to deduce T for arg1 and arg2.

因为编译器在推导arg1和arg2的类型时会遇到类型冲突。

Thus, the effect of class std::reference_wrapper<> is to be able to use a reference as a “first class object,” which you can copy and therefore pass by value to function templates. You can also use it in classes, for example, to hold references to objects in containers. But you always finally need a conversion back to the underlying type.

综上，std::reference_wrapper<>是为了使我们能够像“第一类对象(first class object)”一样地使用引用。可以对其进行拷贝并按值传递给函数模板，也可以在类内部使用它。例如，在容器内部持有对象的引用（译注：stl容器提供的是value语义而不是reference语义，所以容器不支持元素为引用，而用reference_wrapper可以实现）。但是通常最终总是需要将其转换回原始类型。

【编程实验】std::ref的应用

```
#include <iostream>
#include <functional> //for std::cref/std::ref
#include <vector>
#include <list>
#include <numeric>      // std::iota
#include <random>

void f(int& n1, int& n2, const int& n3)
{
    std::cout << "In function:" << n1 << ' ' << n2 << ' ' << n3 << std::endl;
    ++n1; //函数对象的n1副本
    ++n2; //main()中的n2
    //++n3; //编译错误
}

void print_non_template(int x) //
{
    std::cout << "x = " << x << std::endl;
    ++x;
}

template<typename T>
void print_template(T x)
{
    std::cout << "x = " << x << std::endl;
    ++x;
}

template<typename T>
void printV(T s)
{
    std::cout << "s = " << s << std::endl;
}

int main()
{
    /* 1.std::ref的应用：向std::bind/std::thread按引用语义传参*/
    int n1 = 1, n2 = 2, n3 = 3;
    std::function<void()> bound_f = std::bind(f, n1, std::ref(n2), std::cref(n3));
    n1 = 10;
    n2 = 11;
    n3 = 12;

    std::cout << "Before function:" << n1 << ' ' << n2 << ' ' << n3 << std::endl;
    bound_f();
    std::cout << "After function: " << n1 << ' ' << n2 << ' ' << n3 << std::endl;

    /* 2. std::ref/std::cref只能用在模板函数中，在非模板函数中无法达到预期目的*/
    //2.1 非模板函数中使用std::ref传参，结果并不是预期的。无法达到模拟引用传递的目的
    int x = 0;
    print_non_template(std::ref(x)); //std::ref(x)被隐式转换为int&，再按值传入
    std::cout << "x = " << x << std::endl; //x = 0

    //2.2 在模板函数中使用std::ref传参，可以模板引用传递。
    x = 0;
    print_template(std::ref(x));
    std::cout << "x = " << x << std::endl; //x = 1

    std::string s = "hello";
    printV(s); //OK
    printV(std::ref(x)); //OK，在printV中，std::cout的重载版本:basic_ostream& operator<<( int value );
                         //会先尝试将std::reference_wrapper<int>隐式转换为int类型。因此，编译通过。
                         //即std::cout会提示std::reference_wrapper<int>隐式转换为int
    //printV(std::ref(s)); // ERROR: std::reference_wrapper<>类没有重载operator <<。而std::cout重载
                           //中并没有接受string&类型形参的版本，因此std::cout并不知如何转换
                           //std::reference_wrapper，因此编译失败。

    /* 3. std::reference_wrapper与vector */
    //vector是值语义，不支持元素为引用，可用std::reference_wrapper来解决
    std::list<int> ls(10);
    std::iota(ls.begin(), ls.end(), -4); //用连接填充ls，首元素为-4，之后元素依次自增1

    std::cout << "Contents of the list: ";
    for (int item : ls) std::cout << item << " "; std::cout << '\n';

    //vector保存着ls中元素的引用
    std::vector<std::reference_wrapper<int>> vec(ls.begin(), ls.end());

    //打乱vec列表(注意，无法打乱list，因为shuffle需要随机访问)
    std::shuffle(vec.begin(), vec.end(), std::mt19937{ std::random_device{}() });
    std::cout << "Contents of the vec: ";
    for (int item : vec) std::cout << item << " "; std::cout << '\n';

    //将vector中的元素*2;
    std::cout << "Doubling the values in the initial list...\n";
    for (int& item : vec) item *= 2;

    std::cout << "Contents of the list: ";
    for (int item : ls) std::cout << item << " "; std::cout << '\n';

    return 0;
}
/*输出结果
Before function:10 11 12
In function:1 11 12
After function: 10 12 12
x = 0
x = 0
x = 0
x = 1
Contents of the list: -4 -3 -2 -1 0 1 2 3 4 5
Contents of the vec: 4 -3 -2 2 1 3 0 -4 5 -1
Doubling the values in the initial list...
Contents of the list: -8 -6 -4 -2 0 2 4 6 8 10
*/
```