Binder Problem and Reference Proposal

Bjarne Stroustrup (bs@research.att.com)

AT&T Labs
Florham Park, NJ, USA

ABSTRACT

Binders don't work for functions that take reference arguments. The reason is that the bound argument value is stored as a reference. That reference is of type argument to the argument type (which is itself a reference). The suggested solution is to define ***T***&& to mean ***T***&.

1 The Problem

Here is what appears to be an interesting example sent to me by Chuck Allison:

```
#include <algorithm>
#include <iostream>
#include <string>
#include <functional>
using namespace std;

struct Person
{
   string name;
   int year;
   int month;
   int day;

   Person() : name("") { year = month = day = 0; }

   Person(const string& nm, int y, int m, int d) : name(nm) { year = y; month = m; day = d; }
};

bool operator==(const Person& p1, const Person& p2)
{
   return p1.name==p2.name && p1.year==p2.year && p1.month==p2.month && p1.day==p2.day;
}

ostream& operator<<(ostream& os, const Person& p)
{
   os << '{' << p.name << ',' << p.month << '/' << p.day << '/' << p.year << '}';
   return os;
}

bool byName(const Person& p, const string& s)      // note: arguments passed by reference
{
   return p.name == s;
}
```

```
int main()
{
    Person a[] = {
        Person("Albert", 1901,1,20);
        Person("Charles", 1897,3,11);
        Person("Horatio", 1835,12,6);
    };
    int n = sizeof a / sizeof a[0];
    Person* past = a + n;
    Person v("Charles", 1897,3,11);

    Person* p = find_if(a, past, bind2nd(ptr_fun(byName), "Charles")); // error: string&&
    if (p != past)
        cout << "found " << *p << " in position " << p - a << endl;
    else
        cout << "item not found\n";
}
```

This seems like a reasonable thing to do. However, it doesn't compile. The reason is that ***bind2nd***() stores a reference to the argument it needs to bind (in a ***binder2nd***). In the case of ***byName***, that argument is a reference argument so that ***binder2nd***'s constructor tries to create a reference to a reference.

You can get the same compile time error with this simplified ***main***():

```
int main()
{
    bind2nd(ptr_fun(byName), "Chuck");          // error: cannot create const string&&
}
```

The definition of ***binder2nd*** (20.3.6.3, [lib.binder.2nd]) is:

```
template <class Operation>
class binder2nd : public unary_function<typename Operation::first_argument_type,
                                        typename Operation::result_type> {
protected:
    Operation op;
    typename Operation::second_argument_type value;

public:
    binder2nd(const Operation& x, const typename Operation::second_argument_type& y);
    typename Operation::result_type operator()
        (const typename Operation::first_argument_type& x) const;
};
```

The problem is ***binder2nd***()'s argument of type ***Operation::second_argument_type&***. In the case of ***byName***, ***Operation::second_argument_type*** is ***const string&***. Had we managed to create a ***binder2nd***, we would have to face the same problem for operator() 's argument.

We cannot bind an argument of a function taking a reference argument!

2 What To Do

I see three obvious approaches to this problem:

[1] Tell users ''then, just don't do that.'' I don't think this is realistic. Arguments passed by reference – and in particular by ***const*** reference – are common and recommended. Often, a user has no control over the definition of such predicate functions and even less control over (or understanding of) the details of binder implementations. This problem must be solved – the questions are ''how?'', ''when?'', and ''who by?''

[2] Add more binders. Unfortunately, I don't see how we can do that without adding new binder names. To define another (overloaded) version of ***bind2nd***() to cope with reference arguments, we would somehow have to overload or specialize based on the difference between a reference and a non-reference. Adding new names would complicate a user interface that already causes eyes of many average-to-good programmers to glaze over.

[3] Have ***binder2nd*** store a copy of its bound argument. This would change semantics and would

introduce serious memory and run-time overhead in exactly the cases where we recommend using reference argument rather than pass-by-value.

I (clearly) don't find any of these alternatives attractive. Furthermore, the problem will occur in many other contexts where people write function objects.

Consider a more radical/general alternative:

[4] Define ***T***&& to mean ***T***&. This variant of the pointer-to-function rule (*f* means &*f* and *pf*( ) means (**pf*)( )) seems to solve this problems in general. It is also similar to the rule that allows ***const T*** for a ***T*** that is already a ***const*** type.

Does this solution have undesirable side effects? I don't see any.

3 Acknowledgements

Thanks to Chuck Allison for sending me this problem and to Andrew Koening for helping with this analysis and proposal.