Flexible Functors and Binders

Anthony Williams

5th July 2001

Introduction

Functors and Binders are the key elements of Functional Programming, a style of programming supported by the Standard C++ Library. Many of the Standard Library algorithms accept Functors as arguments, and there are several Functors provided in the Standard Library. However, binder support is limited to support for unary functors and binary functors. Also, the type of the bound functor is strongly tied to the type of the original, with no support for default arguments, or arguments to be passed other than by `const` reference. In this article I will describe an alternative set of binders and functor wrappers that are more flexible in the way they are called.

The Standard Library Facilities

An article on Functors and Binders wouldn't be complete without a discussion of the Standard C++ Library support. The Standard Library support consists of 4 main things:

- Functor base classes - `std::unary_function` and `std::binary_function` - which provide typedefs for the return type and argument types of the Functor.
- Function adaptors, which create Functors derived from `std::unary_function` or `std::binary_function` from pointers-to-functions, and pointers-to-member-functions - e.g. `std::pointer_to_unary_function` and `std::mem_fun_ref_t`
- A set of standard Functors, e.g. `std:plus` and `std::unary_negate`. There are also Predicates (Functors with a return type of `bool`), such as `std::less`.
- Binders for binary functions, `std::binder1st` and `std::binder2nd`. These bind the first and second parameters, respectively, of a binary function (which must provide the same typedefs as `std::binary_function`, even if it doesn't derive from it), to specified values, yielding an unary function. The argument provided when these Functors are invoked is then passed as the unbound parameter of the original binary function.

In addition, there are helper functions which assist in the creation of the adaptor template classes and binder template classes, by enabling automatic template type deduction to provide the type of the adapted function, or its arguments. Examples of these helper functions are `std::ptr_fun`, `std::mem_fun` and `std::bind1st`.

There was an introduction to the usage of the Standard Library facilties by Steve Love in Overload 43 [1].

A New Way

The restrictions of the Standard Library support for Functors and Binders can be quite limiting - what if you have a function that takes 3 parameters? What if you want the first and third parameters binding to given values? The Standard Library cannot help you here, so you need a new set of Functor and Binder classes.

Adapting Plain Functions

What do we need in order to wrap a plain function in a Functor class? At the simplest level, let's consider a function that takes no arguments - we need to know its return type, and that is all. What about a function that has all-default arguments? We still need the return type, but we also need to know the full type, so we can keep a pointer to it. A first stab at a Functor wrapper that can support both of these is:

```
template<typename ReturnType,typename FunctionType>
class FunctorAdaptor
{
public:
    FunctorAdaptor(FunctionType fun_):
        fun(fun_)
    {}
    ReturnType operator()() const
    {
        return fun();
    }
private:
    FunctionType fun;
};
```

This class will support *any* Functor that can be called with no arguments, provided we know its return type. The only problem is creating an object - specifying the template parameters can be messy. For this purpose, we provide some helper functions:

```
template<typename ReturnType>
FunctorAdaptor<ReturnType,ReturnType (*)()>
    adapt(ReturnType (*fun)())
{
    return fun;
}
template<typename ReturnType,typename Arg1>
FunctorAdaptor<ReturnType,ReturnType (*)(Arg1)>
    adapt(ReturnType (*fun)(Arg1))
{
    return fun;
}
template<typename ReturnType,typename Arg1,typename Arg2>
FunctorAdaptor<ReturnType,ReturnType (*)(Arg1,Arg2)>
    adapt(ReturnType (*fun)(Arg1,Arg2))
{
    return fun;
}
template<typename ReturnType,typename Arg1,typename Arg2,typename Arg3>
FunctorAdaptor<ReturnType,ReturnType (*)(Arg1,Arg2,Arg3)>
    adapt(ReturnType (*fun)(Arg1,Arg2,Arg3))
{
    return fun;
}
template<typename FunctorType>
FunctorAdaptor<FunctorType::return_type,FunctorType>
    adapt(FunctorType fun)
{
    return fun;
}
```

These adaptors then create Functor instantiations for any plain functions with up to 3 arguments (or more, by simple extension), or any Functor type that provides a `return_type` as a member (possibly a typedef). This then covers all plain functions (up to the limit on the number of arguments we specify), and all Functors derived from `std::unary_function` or `std::binary_function` - this therefore covers all the Standard Library Functors. For completeness we add a typedef `return_type` to our Functor class, so they look just like any other Functor.

```
template<typename ReturnType,typename FunctionType>
class FunctorAdaptor
{
public:
    typedef ReturnType return_type;
// rest as before
};
```

One thing to note is that our `operator()` still works, even if `return_type` is `void`, on a standard conforming compiler. Some major compilers currently available still have not implemented this feature, and so a workaround is required. The library code available to accompany this article shows one such workaround.

Function Arguments

Even though we can create `FunctorAdaptor`s for functions and functors taking arguments, we can currently only call them if they can be called without arguments. This is obviously a bit of a limitation, so we must extend it to cope with arguments. We will start with one argument, as whatever method we choose should be extensible to more than one.

What type should the argument be? Since this is a general-purpose adaptor class, we don't have that information available, as that is encoded in the `FunctionType` template parameter. Thankfully, the compiler comes to our rescue again, this time with its template function type deduction facilities - if we make our overload of `operator()` a template function, then the compiler can deduce the type of the argument from the type of the supplied parameter.

```
template<typename ReturnType,typename FunctionType>
class FunctorAdaptor
{
public:
    template<typename Arg1>
    ReturnType operator()(Arg1 a1) const
    {
        return fun(a1);
    }
// rest as before
};
```

To support multiple arguments, we supply multiple overloads of `operator()`, with different numbers of parameters; because `FunctorAdaptor` is a template class, only those member functions that are actually invoked are compiled, so it doesn't matter if the adapted function can be called with three arguments unless you call the `FunctorAdaptor` with three arguments.

The problem with this is that the compiler will never deduce `Arg1` to be a reference type, so all parameters will be passed by value. Not only can this be expensive in terms of time and resources, but it is not always possible - the class may not have an accessible copy constructor. This also means that the original object cannot be modified, so if the required parameter is a non-`const` reference, the code will fail silently, with the temporary copy being modified instead. The first problem is fixed by passing the parameters by `const` reference, and this will at least cause compile errors when using a function that requires a non-`const` reference. Obviously, this is still not ideal - we need some way of passing non-`const` references. Here we ask for cooperation from our users - if they want a parameter passed by reference, say so - for which purpose we define a holder template, with a helper function.

```
template<typename ReturnType,typename FunctionType>
class FunctorAdaptor
{
public:
    template<typename Arg1>
    ReturnType operator()(const Arg1& a1) const
    {
        return fun(a1);
    }
// rest as before
};
template<typename T>
class RefHolder
{
public:
    RefHolder(T& ref_):
        ref(ref_)
    {}
    operator T&() const
    {
        return ref;
    }
private:
    T& ref;
};
template<typename T>
RefHolder<T> byRef(T& ref)
{
    return ref;
}
```

Then we call our functors like:

```
SomeFunctor(someArg,byRef(someRefArg));
```

Unfortunately, we can't then use this with the Standard algorithms, because they don't know to use `byRef` when a reference argument is wanted. However, we can note that all the Standard Library algorithms require unary or binary functors only, so we can define some more helper classes and helper functions.

```
template<typename Functor>
class UnaryFunctorArgByRef
{
public:
    typedef typename Functor::return_type return_type;
    UnaryFunctorArgByRef(Functor fun_):
        fun(fun_)
    {}
    template<typename Arg1>
    return_type operator()(Arg1& a1) const
    {
        return fun(byRef(a1));
    }
private:
    Functor fun;
};
template<typename Functor>
class BinaryFunctorArgByRefByVal
```

```
{
public:
    typedef typename Functor::return_type return_type;
    BinaryFunctorArgByRef(Functor fun_):
        fun(fun_)
    {}
    template<typename Arg1,typename Arg2>
    return_type operator()(Arg1& a1,const Arg2& a2) const
    {
        return fun(byRef(a1),a2);
    }
private:
    Functor fun;
};
template<typename Functor>
UnaryFunctorArgByRef unaryByRef(Functor fun)
{
    return fun;
}
template<typename Functor>
BinaryFunctorArgByRefByVal binaryByRefByVal(Functor fun)
{
    return fun;
}
```

We also define `binaryByValByRef` and `binaryByRefByRef` for completeness. It is also relatively straightforward to define helpers for more arguments if required, though the number of variations increases exponentially.

Member Functions

Sometimes it is desirable to use member functions as Functors, for example to call a member function on every object in a container. These must then take at least one parameter, which will provide an object on which to call the member function. The member function itself can be stored using a pointer-to-member-function. Since the syntax for calling a member function through a pointer-to-member-function and an object is quite distinct from the syntax for calling a normal function, we will need a new Functor class. At this point, we need to decide how we will be passing our object - by pointer, or by reference. If we choose “by pointer”, smart pointers can easily be used too, but it prohibits applying member functions to objects held by value in containers. Therefore we will take the Standard Library approach of providing both, with the “default” (i.e. least typing) being “by pointer” - `adapt` does “by pointer”, whereas `adaptObjByRef` does “by reference”. Note also that we need separate overloads of `adapt` for `const` and non-`const` member functions.

```
template<typename ReturnType,typename MemberFunctionType>
class MemberFunctorAdaptor
{
public:
    typedef ReturnType return_type;
    MemberFunctorAdaptor(MemberFunctionType fun_):
        fun(fun_)
    {}
    template<typename ObjectPtrType>
    return_type operator()(ObjectPtrType objPtr) const
    {
        return ((*objPtr).*fun)();
    }
    // we will deal with multiple arguments later
private:
    MemberFunctionType fun;
```

```
};
template<typename ReturnType,typename ObjectType>
MemberFunctorAdaptor<ReturnType,ReturnType (ObjectType::*)()>
    adapt(ReturnType (ObjectType::*fun)())
{
    return fun;
}
template<typename ReturnType,typename ObjectType,typename Arg1>
MemberFunctorAdaptor<ReturnType,ReturnType (ObjectType::*)(Arg1) const>
    adapt(ReturnType (ObjectType::*fun)(Arg1) const)
{
    return fun;
}
template<typename ReturnType,typename ObjectType,typename MemberFunctionType>
class MemberFunctorAdaptorRef
{
public:
    typedef ReturnType return_type;
    MemberFunctorAdaptorRef(MemberFunctionType fun_):
        fun(fun_)
    {}
    return_type operator()(ObjectType& obj) const
    {
        return (obj.*fun)();
    }
private:
    MemberFunctionType fun;
};
template<typename ReturnType,typename ObjectType>
MemberFunctorAdaptorRef<ReturnType,ObjectType,ReturnType (ObjectType::*)()>
    adaptObjByRef(ReturnType (ObjectType::*fun)())
{
    return fun;
}
template<typename ReturnType,typename ObjectType,typename Arg1>
MemberFunctorAdaptorRef<ReturnType,const ObjectType,ReturnType (ObjectType::*)(Arg1) const>
    adaptObjByRef(ReturnType (ObjectType::*fun)(Arg1) const)
{
    return fun;
}
```

The helper functions can obviously be extended to support more arguments. The Functor returned from a call to `adapt` can be called with any type as its first argument, provided it supports `operator*`, yielding a type on which it makes sense to apply the given member function. Thus, for example, a derived class pointer could be used if the member function was a base class member, or a smart pointer such as `std::auto_ptr` could be used. On the other hand, the Functor returned from a call to `adaptObjByRef` can only be called with objects that support member functions of the appropriate type - i.e. objects of the correct class, or a class derived from it.

Binders

Now we can create Functors from normal pointers-to-functions and pointers-to-member-functions, and pass any type or number of arguments to them. However, what if we want to, for example, write every item in a container to `std::cout`, given a function that takes two parameters - the object to write, and the stream to write to? `std::for_each` only supports Functors with one argument and besides, we want to output to the same stream in each case. To do this we use a *Binder*, a Functor that calls another functor, fixing (binding) one of its arguments to a specified value, and passing the others through unchanged. Our Binder class needs to know three things - the Functor it is to call, the object to bind, and the

argument number to bind it to. Since the argument number changes the way the underlying Functor is called, we will define a separate class for each different number. This is the way the Standard Library does it. That leaves a set of template classes, each with two template parameters:

```
template<typename FunctorType,typename BoundObjectType>
class Binder1st
{
public:
    typedef typename FunctorType::return_type return_type;
    Binder1st(FunctorType fun_,BoundObjectType obj_):
        fun(fun_),obj(obj_)
    {}
    return_type operator()() const
    {
        return fun(obj);
    }
    template<typename Arg1>
    return_type operator()(const Arg1& a1) const
    {
        return fun(obj,a1);
    }
    // more overloads of operator()
private:
    FunctorType fun;
    BoundObjectType obj;
};
template<typename FunctorType,typename BoundObjectType>
class Binder3rd
{
public:
    typedef typename FunctorType::return_type return_type;
    Binder3rd(FunctorType fun_,BoundObjectType obj_):
        fun(fun_),obj(obj_)
    {}
    // must provide at least two arguments if bound value is to be the third
    template<typename Arg1,typename Arg2>
    return_type operator()(const Arg1& a1,const Arg2& a2) const
    {
        return fun(a1,a2,obj);
    }
    template<typename Arg1,typename Arg2,typename Arg3>
    return_type operator()(const Arg1& a1,const Arg2& a2,const Arg3& a3) const
    {
        return fun(a1,a2,obj,a3);
    }
    // more overloads of operator()
private:
    FunctorType fun;
    BoundObjectType obj;
};
```

We then define some helper functions to deduce the template parameters, which look like the following:

```
template<typename FunctionType,typename BoundObjectType>
Binder1st<FunctionType,const BoundObjectType&>
    bind1st(FunctionType fun,const BoundObjectType& obj)
{
```

```
    return Binder1st<FunctionType,const BoundObjectType&>(fun,obj);
}
```

Note that here we bind the fixed parameter by a `const` reference, since that is how it will be passed to `operator()` of our Functor classes. This option permits us to bind non-`const` references using `byRef` as before, but *only* if the bound functor is being created as a temporary, since the `RefHolder` object will be destructed at the end of the full expression. Therefore, if we want to keep the bound Functor for later use, we will need to define another helper function to keep a copy of the `RefHolder` object in the bound Functor:

```
template<typename FunctionType,typename BoundObjectType>
Binder1st<FunctionType,RefHolder<BoundObjectType> >
    bind1stByRef(FunctionType fun,BoundObjectType& obj)
{
    return Binder1st<FunctionType,RefHolder<BoundObjectType> >(fun,obj);
}
```

Finally, it can be a bit messy to say `bind1st(adapt(&SomeClass::someMemberFunction),someClassPtr)`, so we define overloads of `bind1st` and `bind1stByRef` which automatically do the adapting for plain functions or member functions as well, e.g.:

```
template<typename ReturnType,typename ObjectType,typename Arg1,typename BoundObjectType>
Binder1st<MemberFunctorAdaptor<ReturnType,ReturnType (ObjectType::*)(Arg1)>,
          const BoundObjectType&>
    bind1st(ReturnType (ObjectType::*fun)(Arg1),const BoundObjectType& obj)
{
    return Binder1st<MemberFunctorAdaptor<ReturnType,
                                          ReturnType (ObjectType::*)(Arg1)>,
                     const BoundObjectType&>(fun,obj);
}
template<typename ReturnType,typename Arg1,typename BoundObjectType>
Binder1st<FunctorAdaptor<ReturnType,ReturnType (*)(Arg1)>,
          RefHolder<BoundObjectType> >
    bind1stByRef(ReturnType (*fun)(Arg1),BoundObjectType& obj)
{
    return Binder1st<FunctorAdaptor<ReturnType,
                                    ReturnType (*)(Arg1)>,
                     RefHolder<BoundObjectType> >(fun,obj);
}
```

Now we have a solution to our original problem of printing the contents of a container:

```
void write(const MyClass&, std::ostream&);
std::for_each(container.begin(),container.end(),
              bind2nd(write,byRef(std::cout)));
```

Call Me Later

Looking at the variety of different types of Functor, with all their different template parameters, creating an object of the correct type is not going to be easy. The user would have to perform the same template type deductions as the compiler, and explicitly specify all the template parameters. We want to avoid this, as it is typing-intensive and error-prone, which is why we provide the helper functions. This is fine for passing the functors to Standard Library algorithms, as they are template functions and thus the type of the Functor is automatically deduced by the compiler, but not for keeping the Functor for later use.

Instead, what we can do is create a wrapper class, for which the user specifies in a straightforward fashion what parameters and return type they require. We then make this wrapper capable of holding any of the miriad of Functors, but only capable of calling them as specified. The user can then pass round this wrapper. Obviously, this is at the expense of a layer of indirection, and so the resultant function call is slower. In this case, the indirection takes the form of a virtual function call - we define a template class derived from a non-template base class. The template class can then be parameterized by the actual Functor type without affecting the containing class - we use a templated constructor of our containing class to pick the right implementation class. We pass all the arguments using `byRef` to ensure that the reference nature is preserved. In order to ensure that the Functor can be freely copied with harm, the polymorphic implementation class is `clone`d when the outer class is copied.

```
template<typename ResultType,typename Arg1,typename Arg2,typename Arg3>
class Functor3
{
private:
    class FunctorImplBase
    {
    public:
        virtual ReturnType call(Arg1 a1,Arg2 a2,Arg3 a3) const=0;
        virtual std::auto_ptr<FunctorImplBase> clone() const=0;
        virtual ~FunctorImplBase()
        {}
    };
    template<typename FunctorType>
    class FunctorImpl:
        public FunctorImplBase
    {
    public:
        FunctorImpl(FunctorType fun_):
            fun(fun_)
        {}
        virtual std::auto_ptr<FunctorImplBase> clone() const
        {
            return std::auto_ptr<FunctorImplBase>(new FunctorImpl<FunctorType>(*this));
        }
        virtual ReturnType call(Arg1 a1,Arg2 a2,Arg3 a3) const
        {
            return fun(byRef(a1),byRef(a2),byRef(a3));
        }
    private:
        FunctorType fun;
    };
public:
    // make this Functor compatible with our binders
    typedef ReturnType return_type;
    // copy constructor - clone the contained Functor
    Functor3(const Functor3& old):
        funPtr(old.funPtr->clone())
    {}
    // template constructor
    template<typename FunctorType>
    Functor3(FunctorType fun):
        funPtr(new FunctorImpl<FunctorType>(fun))
    {}
    ReturnType operator()(Arg1 a1,Arg2 a2,Arg3 a3) const
    {
        return funPtr->call(a1,a2,a3);
    }
private:
```

```
    std::auto_ptr<FunctorImplBase> funPtr;
};
```

We can then keep a copy of our functors for later use, like so:

```
int someFunc(std::string& a,double b); // our functor
// bind a parameter and store it - note the argument and
// return types can differ from the original, provided
// there is an implicit conversion
std::string s("hello");
Functor1<float,int> f(bind1stByRef(someFunc,s));
// now use it
float a=f(19);
```

The resultant `Functor1` object can be used with the standard library algorithms, or with the binders described above, just like any other Functor. It could also, for example, be stored in a container along with other `Functor1` objects witht the same signature, even if the underlying Functor was completely different. Obviously, if an argument is bound by reference, then the Functor is useless if it outlives the referred-to object.

Conclusion

I have introduced a new way of providing Functor and Binder support, which allows for variable numbers of arguments. This is more flexible than the Standard Library facilities in that it supports more than 2 arguments, and Functors with variable number of arguments (e.g. by having parameters with default values). It also simplifies the syntax for binding e.g. member functions to objects, and yet remains compatible with the Standard Library algorithms.

The source code to accompany this article includes support for functors taking up to 15 arguments, and is available from my website.

References and Further Reading

1. *Are You Afraid Of The Dark? - Making Sense of the STL*, Steve Love, Overload 43.
2. *Modern C++ Design*, Andrei Alexandrescu. Chapter 5.
3. *http://cplusplus.anthonyw.cjb.net* - source code to accompany this article, and other articles by the same author.