Interfacing C++ member functions with C libraries

Kurt Vanmechelen and Jan Broeckhove

Department of Mathematics and Computer Sciences
University of Antwerp
B-2020 Antwerp, Belgium
{kurt.vanmechelen, jan.broeckhove}@ua.ac.be

Abstract. Interfacing functors or member functions with C libraries proves to be difficult as library routines can only accept a pointer-to-function as a callback argument. Usually this limitation is addressed by constructing an ad hoc wrapper, but this approach has several drawbacks. We propose a more flexible and generic solution to the problem of mapping functor or object-member function pairs to plain C functions using recursive template programming techniques. A performance analysis of our solution is presented.

1 Introduction

In object-oriented C++ programming object-and-member-function pairs and functors occur as callable entities, besides C-style functions and function pointers [1]. Functors are simply classes that define the call operator, i.e. `operator()`, as a member function. In this contribution, we will use a class *Particle* with members *position* and *velocity* to illustrate our ideas.

```
class Particle {
public:
    double position(double time);
    double velocity(double time);
};
```

When two software components are developed independently, e.g. one's own code and a numerical library from a third party, they are often tied together through the *callback mechanism.* Consider for instance the computation of a particle's acceleration using the library's derivative procedure on the particle's velocity. When the library procedure, referred to as *caller*, executes, it invokes the function whose derivative must be computed. This function is the *callee*, and it is passed to the caller by way of the *callback function* argument. Thus the design of the caller also prescribes the type of the callee that it accepts. There are several approaches for the design of C++ callback libraries [2] [3] which support flexible callback constructs. They are however only applicable when both caller and callee are designed in an object-oriented fashion.

We want to look at the situation that arises when the caller is part of legacy C code. This is a common situation because few developers care to reengineer their existing code to C++. The question arises as to how C++ functors and member functions can be hooked into C-style callbacks. When the caller is a C procedure, as illustrated below, the callee is necessarily a pointer to function, with a type determined by its signature (argument types and the return type).

```
double derivative(double step, double x, double (*f)(double));
```

On the face of it, the member function in `Particle::velocity` has the appropriate signature, suggesting that its address can be used as callback function argument. However, a member function needs to be bound to an object instance at runtime in order to make sense, an implicit "this" pointer in its argument list points to that object. As a consequence, the member function signature is not compatible with the pointer-to-function accepted by a C-style callback.

In this contribution we want to develop a mechanism that enables object-oriented code to interface with C libraries when the connection must be made via the C-style callback. Our approach extends previous work on this mechanism for functors [4]. In the following sections we first consider the most commonly used approach, that of the ad hoc wrapper, and its limits and drawbacks. These are addressed in our approach of an adapter, which we generate through recursive template instantiation techniques. We will also present a performance analysis of our solution.

2 The ad hoc wrapper approach

This solution uses a static member function to bind the callee to the caller. An invocation of a static member function does not require a "this" pointer to provide a calling context. Pointers to such static member functions are therefore convertible to C-style function pointers that host the same method signature.

```
class ParticleWrapper {
public:
    static double velocityGlue(double time)
                       { return fObj->velocity(time); }
    static void setObj(Particle& obj) { fObj = &obj; }
private:
    static Particle* fObj;
};

// Bind the wrapper to particle p and call procedure
ParticleWrapper::setObj(p);
double res = derivative(0.001,2.0,&ParticleWrapper::velocityGlue);
```

A static data member holds the reference to the callee object that will receive the adapted call. Static functions need to be defined in the wrapper for every member function that is to be adapted.

This approach is the most common way of dealing with the problem of binding member functions to C-style callbacks [5]. However, it suffers from a number of limitations. Firstly, one needs to manually define the necessary wrappers for every class and for every member function that requires adaptation. Secondly, one can only adapt a member function for a single callee instance at a time because there's only one static data member to hold the instance's address. Indeed, when a second object is adapted, one overwrites the previous object's address. If callers have stored the function pointer for continued use, then calling that pointer now invokes the member function on the second object instead of on the first.

3 The adapter approach

This section presents the design of a generic *adapter* class that is able to bind member functions to C-style function pointers for a number of callee objects. The number of callee objects that can be adapted will need to be specified at compile-time. The core of the approach is again to use static functions to glue the member function implementations to a calling interface that is convertible to C-style function pointers. However, the adapter will support objects and member functions of arbitrary type and signature, alleviating the first limitation of the ad hoc approach. Furthermore, our solution will be able to support the adaptation of multiple callee objects of the same type, at the same time, thereby addressing the ad hoc wrapper's second limitation.

In order to support the adaptation of multiple callee instances, we introduce a mapping structure that maps pairs of object/member function addresses to associated glue functions.

```
template <class KeyType, class MappedType, int mapCapacity>
class IndexedMap : public vector<pair<KeyType, MappedType> >
```

When a member function is adapted for the first time, a key-value pair is added to the map. The addresses of the callee object and its member function serve as the key, with the address of an available glue function as its associated value. The map is wrapped inside Loki's `SingletonHolder` (cfr. [6] for the Loki library) which creates and holds a unique instance of the type defined by its template parameter.

The glue functions retrieve a callee object's address at a fixed position in the map and invoke the appropriate member function. A glue function is wrapped inside a template class to support arbitrary callee types and member function signatures. A specialization [7] for a wrapper class that supports member functions accepting a single argument is shown below.

```
template <class CTraits, int mapMax, int i> class Wrapper {};

//Specialization of the Wrapper template for member functions
//accepting one argument
template<class ObjectType, class R, class P1, int MapMax, int i>
class Wrapper<MemberFunctionTraits<ObjectType, R, TYPELIST_1(P1)>,
                                   MapMax, i>
{
public:
    //The C-Style function pointer type
    typedef R (*FP)(P1);

    //The KeyType for indexing the map
    typedef pair< ObjectType*,
                  MemberFunctionTraits<ObjectType, R, TYPELIST_1(P1)
                  >::MemberFunctionPointerType > MemFuncKeyType;

    //The IndexedMap singleton
    typedef SingletonHolder< IndexedMap<MemFuncKeyType, FP, MapMax>,
                             CreateStatic, NoDestroy > A2FMap;

    //The forwarding function
    static R forwardCall(P1 parm1)
    {
        MemFuncKeyType key = (A2FMap::Instance())[i].first;
        return (key.first->*key.second)(parm1);
    }
};
```

We use the *traits* [8] technique to combine all information concerning the member function's type in the `MemberFunctionTraits` class. Encapsulation of type information within a traits class increases the modularity and resulting extensibility of the template structure. The function's argument types are passed to the traits class through a `TYPELIST` construct provided by the Loki library. A `TYPELIST` is a container for types. Loki provides operations in the form of template classes to manipulate the list at compile-time.

In order to support the adaptation of the same member function for n instances of the callee class, we need to generate n static glue functions. We add an extra `int` template parameter to the wrapper class that hosts the static glue function for this purpose. The integer parameter will denote the index of the object/member function pair that will be adapted by the glue function. Every time the compiler instantiates the `Wrapper` class with a new value for i, a static glue function will be generated. To perform the instantiation process, we use the recursive template algorithm shown below.

```
template<class CT, template<class,int,int> class Glue,
         int mapMax, int i>
class GlueList {
public:
    //The typelist of the previous GlueList instantiation
    typedef GlueList<CT, Glue, mapMax, i-1>::typeList pList;

    //Append a new wrapper class instantiation to the typelist
    typedef Glue<CT, mapMax, i> newGlue;
    typedef typename Append<pList, newGlue>::Result typeList;
};

//Specialization representing the base case for the recursion
template<class CT, template<class,int,int> class Glue, int mapMax>
class GlueList<CT, Glue, mapMax, 0> {
public:
    typedef Glue<CT, mapMax, 0> newGlue;
    typedef TYPELIST_1(newGlue) typeList;
};
```

The i parameter specifies the number of `Glue` class instantiations that need to be made. The `GlueList` class defines a publicly available `typeList` type. At the end of the recursion, this typelist will contain all the `Glue` instantiations. In every step of the algorithm we take the list of the $i-1$'th `GlueList` and append a new instantiation of `Glue` to it. The compiler continues the recursive instantiation process until i reaches 0. At this point, the specialization of the `GlueList` template for $i = 0$ is instantiated and the recursion ends.

The glue function addresses of these wrapper classes are inserted into the `IndexedMap` singleton by means of a type-iterative algorithm based on recursive template instantiation (no code shown). The algorithm iterates over the typelist constructed by the `GlueList` template. In every step of the recursion, the address of the glue function belonging to the wrapper class at the head of the list is inserted into the map. Recursion continues until the tail of the typelist equals `NullType`, indicating the end of the list.

The code fragment below demonstrates the use of our final solution by adapting the member function `velocity` of the `Particle` class defined in the introductory section. The adapted member function is then passed on as a pointer-to-function argument of the `derivative` function contained in a C library.

```
//Define a 10-slot adapter and get the instance
typedef Adapter<Particle,double,TYPELIST_1(double),10> PAdapter;
PAdapter* ad = &PAdapter::Instance();

//Adapt a particle p's velocity function
PAdapter::FunctionPointerType fp=ad->adapt(p, &Particle::velocity);
double res = derivative(0.001, 2, fp);
```

4 Performance Evaluation

Our adapter provides a more generic and flexible solution to the member function adaptation problem. This section will determine the associated cost of this flexibility by comparing the performance of C callbacks using the adhoc wrapper approach versus callbacks using our adapter.

Measurements were obtained on a 2.4 GHz Pentium IV processor with 512 Kb L2 cache and 512 Mb of RAM. The adapter has been compiled and tested on the following platforms; gcc 3.2.2 and 3.3 on Solaris and SuSE Linux, Comeau 4.3 with a SunONE CC 5.1 backend on Solaris, Intel C++ 7.1 on Windows XP and SuSe Linux, Microsoft Visual 2003 C++ 7.1 and Metrowerks C++ 8.3 on Windows XP. All tests ran under a thread with critical priority. In this section, we present timings for the Visual 7.1, Intel 7.1 and gcc 3.3 compilers.

Our test setup consists of a C library function that calls back to a member function which returns the sum of two integers. We will measure the time it takes for the library function to return, i.e member function execution time is included in the measurements. In order to prevent cross-source compiler optimizations, we compiled the library source separately using the highest optimization level. We enabled automatic inlining for all compilers and optimization levels.

Intel's RDTSC [9] instruction was used to measure the execution time of the library function. The RDTSC assembly instruction returns the current 64 bit value of the Pentium's TSC (Time Stamp Counter). The TSC is reset on boot and increments every clockcycle. RDTSC reads the low-order 32 bits of the TSC into the accumulator. The RDTSC instruction does not qualify as a serializing instruction. Therefore, it may be executed out of order with respect to instructions preceding or following it. To prevent this, we issued a CPUID instruction before every call to RDTSC. CPUID returns information about the CPU and is the only serializing instruction callable from user mode. The overhead for issuing the RDTSC/CPUID instruction pair was subtracted from the measured result. The library function was called ten times. The first call includes all main memory transfer times and cache miss overhead, it serves as a warmup. We took the minimum of the other nine calls to denote the minimal execution time of the library function.

Table 1 shows the values of these measurements for different compilers, platforms and optimization levels. For the Visual compiler, the extra cost of using our adapter is 12 cycles on the highest optimization level. Object code produced by the gcc compiler shows slightly higher execution times for both ad hoc and adapted cases. The extra overhead incurred by our adapter results from accessing the map structure, but more importantly, from the fact that the code for the forwarded member function did not get inlined in the glue function body, in contrast to the ad hoc case. This was determined by inspecting the generated assembly code. The table also shows that the impact of the extra statements in the adapter's wrapper function is heavily reduced by the compiler's optimizations. The OS has a small impact on the code's performance as shown by the measurements for the Intel compiler on SuSe versus those on Windows XP.

Table 1. Time per callback in clock cycles for the ad hoc case and adapted case on different compilers, platforms and optimization levels.

	VC 7.1 XP			Intel 7.1 XP				Intel 7.1 SuSe				gcc 3.3 SuSe			
Optimization	O2	O1	O0	O3	O2	O1	O0	O3	O2	O1	O0	O3	O2	O1	O0
Adapted	28	32	72	32	32	92	184	36	36	36	100	40	48	64	164
Ad hoc	16	16	16	16	16	16	32	16	16	16	40	20	20	20	40
Overhead	12	16	56	16	16	76	152	20	20	20	60	20	28	44	124

Previous work [4] using the same test setup, showed a smaller overhead of 10% for the Visual compiler when adapting a functor's call operator. In the functor case, the operator call was hard coded into the wrapper's glue function, which enabled the compiler to inline its code.

5 Conclusion

A flexible solution was presented to tackle the problem of adapting member functions to C-style function pointers. In contrast to the ad hoc wrapper solution, it is possible to adapt multiple object and member function types. The number of object/member function pairs that can be adapted is tunable at compile time on a type to type basis. Performance analysis has quantified the overhead of our solution compared to the ad hoc approach, and has shown the effect of compiler optimizations in this regard.

References

1. J. Barton and L. Nackman. *Scientific and engineering C++.* Addison-Wesley, 1994.
2. R. Hickey. Callbacks in C++ Using Template Functors. *C++ Report*, 7(2), pages 42-50, February 1995.
3. P. Jakubic. Callback Implementations in C++. Proceedings of the 23rd Technology of Object-Oriented Languages Conference, TOOLS-23, pages 377-406, Eds. IEEE Computer Society Press., Santa Barbara, CA, USA, July 1997.
4. J. Broeckhove and K. Vanmechelen. *Using C++ functors with legacy C libraries.* Proceedings of the 2004 International Conference on Computational Science and its Applications, ICCSA 2004, number 3046 in Lecture Notes in Computer Science, pages 514-523, Assisi, Italy, May 2004.
5. L. Haendel. The function pointer tutorials. http://www.function-pointer.org.
6. A. Alexandrescu. *Modern C++ Design.* Addison-Wesly, 2001.
7. D. Vandevoorde and N. Josuttis. *C++ templates.* Pearson Education, 2003.
8. N. Meyers. Traits: A New and Useful Template Technique. *C++ Report*, 7(5), pages 32-35, June 1995.
9. Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z.* Intel Corporation, 2004.