An Adaptor for C++ callbacks with C and Fortran Libraries

J. Broeckhove, K. Vanmechelen

Department of Mathematics and Computer Science, Universiteit Antwerpen, Middelheimlaan 1, Antwerpen 2020, BE

Abstract

Object-oriented programming using C++ is increasingly being adopted in the development of scientific codes. A recurrent issue in this regard, is the interaction of newly developed codes with existing legacy libraries written in C or Fortran. Often, one needs to pass raw function pointers to such libraries’ procedures for callback purposes. This is problematic as it conflicts with one of the cornerstones of object-oriented programming, the association of functions and data through objects. Currently ad-hoc approaches are used to deal with this issue, but these are error-prone and lack reusability. We present a generic adaptor that is able to wrap any callable C++ entity and provide a raw function pointer that is compatible with C or Fortran library routines. This allows for an object-oriented style of programming, while interfacing with legacy libraries in a straightforward manner.

Key words: C++ callbacks, C libraries, Fortran libraries, C++ template programming

PACS: P89.20.Ff; 07.05.Bx

PROGRAM SUMMARY

Manuscript Title: An Adaptor for C++ callbacks with C and Fortran Libraries
Authors: J. Broeckhove and K. Vanmechelen
Program Title: Adapt2rfp
Journal Reference:
Catalogue identifier:
Licensing provisions: none
Programming language: C++
Computer: All
Operating system: All
Keywords: C++ callbacks, C libraries, Fortran libraries, C++ template programming
PACS: P89.20.Ff, 07.05.Bx
Classification: 6.5, 4.14

Email addresses: jan.broeckhove@ua.ac.be (J. Broeckhove), kurt.vanmechelen@ua.ac.be (K. Vanmechelen)
**Nature of problem:**
Object-oriented programming using C++ is increasingly being adopted in the development of scientific codes. A recurrent issue in this regard, is the interaction of newly developed codes with existing legacy libraries written in C or Fortran. Often, one needs to pass raw function pointers to such libraries’ procedures for callback purposes. This is problematic as it conflicts with one of the cornerstones of object-oriented programming, the association of functions and data through objects. Currently ad-hoc approaches are used to deal with this issue, but these are error-prone and lack reusability.

**Solution method:**
Recursive template instantiation is used to generate instantiations of wrapper templates. These template classes provide a static forwarding function that can be converted to a raw function pointer. The necessary provisions are in place to deal with variations in call signatures.

**Restrictions:**
The present adaptor implementation can handle callable entities with signatures of at most nine parameters. Other implementations supporting more parameters can be generated but require the Boost macro library. The code of the adaptor implementation (a single header file) fails to compile on compilers pre-dating the introduction of TR1 C++ library extension. E.g. for the gcc suite one needs version 4.3 (released early 2008) or above.

**Unusual features:**
The inclusion of a single header file `adapt2rfp.h` suffices to integrate the solution in an existing software project.

**Running time:**
A call through a raw function pointer returned by the adaptor adds on the order of 30 machine instructions to forward the call through the adaptor’s wrapper structure. Due to the static nature of these forwarding calls, the instruction count can be heavily optimized by the compiler.
1. Introduction

An object-oriented style of programming using C++ is definitely a sensible option in present-day scientific software development [1]. However, the effort involved in recoding legacy libraries from C or Fortran to C++ is prohibitive. A need therefore arises to interface newly developed C++ code with these legacy libraries.

Libraries often interact with the client code through callbacks. For instance, when an application requires the root of a function and calls a numerical library to compute it, it needs to supply that function to the library. Usually this is achieved by introducing a pointer-to-function argument in the library procedure’s parameter list. The pointer-to-function callback allows the library to call back in the application code to evaluate function values. Libraries for scientific computing that intensively use such callbacks include for example the NUMAL library [2], the ISML library [3] and libraries hosted at the Netlib repository such as the rksuite library for solving Initial Value Problems using Runge-Kutta methods [4].

The use of a pointer-to-function callback in a library procedure’s parameter list does not present any problems when the entity to be called in the C++ application code is a freestanding (i.e. non-member) function. The extern "C" linkage specification can be used to enable linking between the C++ application and the C or Fortran library.

When functors or member functions are to be called however, the signature of the library procedure – which is fixed – does not allow for both an object pointer and a member function pointer to be passed. This problem is usually addressed in an ad-hoc manner. One defines global variables to store those object and member function pointers and introduces a freestanding function that refers to them to forward the call to the member function acting on the object. A pointer to this ad-hoc freestanding function is then passed as the callback argument. A somewhat cleaner approach consists in defining an ad-hoc wrapper class that holds the pointers in static data members and that has a static member function to forward the call. A pointer to this static member function then acts as the callback.

This ad-hoc solution is not optimal for a number of reasons. First, there is no reuse of the wrapper code: every time a callback with a new signature occurs, new wrapper code needs to be written. Second, if the library procedure has multiple callbacks of the same signature, as is the case for many differential equation solvers for example, multiple wrapper classes need to be written. Third, a potential for coding errors is introduced. When library calls that involve callback arguments occur at different points in the application code, one must guarantee that the wrapper class holds the appropriate pointers to service the call correctly. At the expense of additional plumbing code, one could add the necessary checks or re-initialize the wrapper prior to the library procedure call. However, if the library stores the callback argument for later use, this...
approach is insufficient and one is forced to write a separate wrapper class for every call to the library.

The function call wrapper in the C++11 Standard Library (std::function, [5], [6]) or its analog in the Boost Library ([7]) address part of the problem we face. It provides an entity with a single interface that can hold different types of callable entities, and thereby improves reusability. However, that in itself does not make it possible to associate with it a pointer-to-function for the callback. For that, construction of a wrapper with a static member function is still necessary.

We propose a solution that incorporates the use of the std::function and uses C++ templates to automatically generate wrapper classes for any C++ callable entity. Our solution relies on the recursive instantiation of C++ templates for the compile-time generation of wrappers. The only limitation of our solution is that the maximum number of wrappers that can exist throughout the program’s execution is fixed by a compile-time constant specified by the user code, and that it can handle only callback with signatures of at most ten parameters.

2. Theoretical background

We first refresh some C++ terminology [5] to be precise in the subsequent discussion.

Callable types are types that support a call operation such as a freestanding function, pointer to freestanding function, functor, pointer to member function, etc. A callable object is an instance of a callable type (the object in this term does not imply that it is an instance of a class, e.g. a freestanding function is a callable object).

A call wrapper type holds a callable object and applies calls through that object. A call wrapper is an instance of a call wrapper type. The callable object held by the wrapper is known as the target object.

A function type describes the return type and argument types in a syntax construct that consists of the return type followed by a left parenthesis, a (possibly empty) comma-separated list of argument types and a right parenthesis. For instance, the function type of \texttt{exp} in Listing 1 is double (double).

```c
// C/C++ header declaring C library quadrature procedure
#ifdef _cplusplus
extern "C" {
#endif
    double quad(double a, double b, double tol, double (*f)(double));
#ifdef _cplusplus
}
#endif

// application code function to be used as callback
double exp(double x);

// application code function to be used as callback
float expf(float x);
```
2.1. Callbacks with C or Fortran Libraries

As a running example, we consider a quadrature procedure `quad`, available in a C library. Typically it has a prototype such as in Listing 1 where `a` and `b` are the interval boundaries and `tol` the tolerance used in the algorithm. The last parameter is referred to as a callback because it allows the library procedure to call into the application code, in this case to evaluate values of the function whose quadrature is to be computed. The library header contains an `extern "C"` qualification, that is not active when the library is compiled by the C compiler but that is active when the application code is processed by the C++ compiler. It specifies that the linkage has to follow C calling conventions. The client application defines a function, the exponential function `exp` in the example of Listing 1, and calls the library procedure. The callback argument is automatically converted to a pointer, but one can also explicitly use `&exp` as an argument.

For Fortran libraries the setting is analogous. An `extern "C"` declaration is required to specify correct linkage to the Fortran library routines. In this linkage the standard rules for translating Fortran arguments into arguments of the corresponding C function apply. That is, an argument of type `external` in the Fortran function which represents a callback, translates to a raw function pointer in C/C++. In the type of the raw function pointer, all of the function’s formal arguments are of pointer type. Name mangling by the Fortran compiler might require appending an underscore to the names of the library’s procedures when exposing them in a C/C++ header. The name mangling is compiler dependent.

```
// Functor raises argument to fixed power, set through constructor.
class Pow {
  public:
    Pow(double exponent) : fExponent(exponent) {}
    double operator()(double x) { return pow(x, fExponent); }
    double deriv(double x) { return fExponent * (x + pow(x, fExponent) - 1.0); }
  private:
    double fExponent;
};
```

Listing 2: Definition of the `Pow` functor (using the Standard Library’s `pow` function).

2.2. Callbacks with C++ Libraries

For C++ application code, the above scheme is rigid in two respects. Firstly, the function type of the callback is fixed. Function types that are compatible through conversion of return and/or argument types cannot be used. For example, in Listing 1, an invocation of `quad` with `expf` as an argument for the callback
parameter fails to compile. Secondly, the callable type of the callback is limited to a (non-member) function or pointer to function. Callable types that are specific to an object-oriented programming style such as functor classes or pointers to member functions, as shown in Listing 2, cannot be used. For instance with the code in Listing 2, one cannot apply the quad procedure to an instantiated Pow functor or to its deriv member function using the \&Pow::deriv pointer.

Neither of these restrictions apply to modern C++ library interfaces that are constructed with the std::function call wrapper. The std::function type, now part of the C++ Standard Library, was introduced in the extension commonly known as Technical Report 1 (TR1) or officially ISO/IEC TR 19768, C++ Library Extension, the document detailing the extension. In this case, the quad procedure’s function type can be defined as in Listing 3, with the formal callback parameter of type std::function.

\[
\text{double quad}(\text{double a, double b, double tol, function\langle\text{double(double)}\rangle f);}
\]

... 
\[
y = \text{quad}(0., 1., .01, \exp);
\]

... 
\[
\text{Pow p3}(3.0);
y = \text{quad}(0., 1., .01, \expf);
\]

Listing 3: Code fragments showing a callback scheme with a C++ library

The call wrapper object \texttt{f} can hold a target object of any callable type with argument type convertible from double and return type convertible to double. The \texttt{bind1st} and \texttt{mem\_fun} functions of the Standard Library are used to bind the functor’s deriv member function pointer to the \texttt{p3} object.

In Listing 3, the first two invocations of \texttt{quad} demonstrate the flexibility with respect to the compatibility of argument and return type of the callback, while the next two invocations demonstrate the flexibility with respect to the callable type.

\[
\text{struct AdHocPow} \\
\text{static Pow fTarget;}
\text{static double forwarder(double x) \{ return fTarget(x); \}}
\]

... 
\[
\text{extern "C" double quad(double a, double b, double tol, double f(double));}
\]

... 
\[
\text{Pow p3}(3.0);\text{AdHocPow fTarget = p3;}
y = \text{quad}(0., 1., .01, \text{AdHocPow.forwarder});
\]

Listing 4: C++ Code fragments illustrating the use of an ad-hoc adaptor

2.3. C++ application code using C or Fortran libraries

When C++ application code has to use a C or Fortran library, one is faced with a rigid interface as in Listing 1. Such interfaces can also be found in some C++ libraries, either for the purpose of backward compatibility or because they were developed before the introduction of the function call wrapper.
In C++ discussion groups, the question of using an *adaptor* to transform a functor call into a free function call for the purpose of a callback, crops up quite often and is frequently answered by an ad-hoc construct as in Listing 4.

The adaptor is itself a call wrapper that stores the functor and allows it to be called through a static member function. The latter does not require a hidden object pointer argument and is thus equivalent to a freestanding function. It can therefore be used as a callback argument. Because the target object is accessed in the static member function, it also needs to be static.

Clearly, the adaptor can be made more versatile and reusable by using the `std::function` type in its construction. This opens up possibilities outlined previously for C++ applications as demonstrated in Listing 5. Again we have flexibility with respect to compatible function types and callable types of the callback argument.

```cpp
struct AdHoc {
    static function<double(double)> fTarget;
    static double forwarder(double x) { return fTarget(x); }
}

... extern "C" double quad(double a, double b, double tol, double (f)(double));

AdHoc.fTarget = exp;
y = quad(0., 1., .01, AdHoc.forwarder);

AdHoc.fTarget = expf;
y = quad(0., 1., .01, AdHoc.forwarder);

AdHoc.fTarget = p3;
y = quad(0., 1., .01, AdHoc.forwarder);

AdHoc.fTarget = std::bind1st(std::mem_fun(&Pow::deriv), &p3);
y = quad(0., 1., .01, AdHoc.forwarder);
```

Listing 5: Illustration the ad-hoc adaptor (refer to previous listings for some definitions).

Listing 5 shows a float version `expf` wrapped in a `function<double(double)>` (let us assume that a double precision `exp` is not available) and the ad-hoc wrapper provides the callback with the `double(double)` signature required by the `quad` procedure. This does not of course enhance the `expf` implementation to double precision. As always, one needs to be aware of the conversions being effected and their potential impact on numerical precision.

With the ad-hoc approach, one is forced to explicitly write an adaptor class for every function prototype of the callback formal parameters that occurs. This is because the function type of `AdHoc::forwarder` needs to be identical to that of the callback argument. Changing the return type of the callback argument to `float` in Listing 5, requires recoding the ad-hoc adaptor. Or, if multiple callbacks of different function type are needed, multiple wrappers need to be coded.

```cpp
template<typename S> struct AdHoc;
template<typename R> struct AdHoc<R>() {
    static function<R()> fTarget;
    static R forwarder() { return fTarget(); }
}
```

7
template<typename R, typename P1> struct AdHoc<R(P1)> {  
    static function<R(P1)> fTarget;
    static R forwarder(P1 p1) { return fTarget(p1); }  
}
template<typename R, typename P1, typename P2> struct AdHoc<R(P1, P2)> {  
    static function<R(P1, P2)> fTarget;
    static R forwarder(P1 p1, P2 p2) { return fTarget(p1, p2); }  
}
...  
extern "C" double quad(double a, double b, double tol, double (*f)(double));
extern "C" int make_table(int max, double (*f)(double, double));
...
AdHoc<double(double)> fTarget = exp;
y = quad(0., 1., .01, AdHoc<double(double)>::forwarder);
AdHoc<double(double, double)> fTarget = pow;
make_table(2, AdHoc<double(double, double)>::forwarder);

Listing 6: C++ code fragments illustrating the use of the templated ad-hoc adaptor

This lack of code reuse is cumbersome and can be avoided by turning the adaptor class into a template, as in Listing 6. One declares a primary template to introduce the template name to the compiler, and provides template specializations for the nullary, unary, binary, etc., cases. These specializations are themselves templated with respect to the return type and parameter types of the static member function of the adaptor. If one provides specializations up to N parameters, one can deal with function types of up to N parameters.

Listing 6 illustrates the approach. The invocation of the quad procedure is handled with the unary specialization, whereas the binary specialization handles the instantiation in the make_table call.

The templated ad-hoc adaptor construction still has serious shortcomings. For instance, the procedure femlagskew in the NUMAL C library [2, p472] solves second-order, linear, self-adjoint two-point boundary problems familiar from Sturm-Liouville theory. It has the prototype indicated in Listing 7. The three callbacks correspond to functions occurring in the Sturm-Liouville differential equation. In a situation like this, the approach of Listing 6 requires one to write three templated adaptors and specializations. This is because the adaptor members are necessarily static and thus three distinct instances each having a distinct target data member are required.

Another shortcoming relates to managing the lifetime of the adaptors. The situation of Listing 5 is representative of most general-purpose libraries. Once the library procedure finishes and returns to the application code, the state of the callback becomes irrelevant. This means that the wrapper can be re-assigned for use in a subsequent call to a library procedure. However, when using frameworks, the situation is different. The shift of control from user code to the framework, means that one has to initialize an execution context in the framework by registering callbacks with it. Procedures in the framework obtain the callbacks, when needed, from that execution context. This simplifies the
interface to the framework and guarantees the integrity of the execution context.
The latter aspect is hard to manage if reuse of the adaptor is possible, precisely
because its members are static. When another callable object is assigned to the
adaptor, the application continues to execute but the integrity of the execution
context in the framework and hence the correctness of the code are compromised.

There is no straightforward object-oriented solution in the spirit of “resource
acquisition is initialization” (RAII) [8, p366] to the problem raised in the pre-
vious paragraphs. The resource in question, the static data member holding
the callable object and the static member function of the wrapper, cannot be
created and destroyed as an object. They are static and must be generated at
compile-time. We therefore need a means of generating, at compile time, an
arbitrary number of wrapper classes of arbitrary function type and need a policy
concerning their access and reuse.

Yet another limitation, particularly relevant when interfacing with Fortran
libraries, lies in the function type of the callback. Fortran uses “call by address”
argument types only and consequently all argument types in the callback are
pointer types. As an example, consider the well known QUADPACK quadrature
package [9] available at the Netlib software repository [10]. The dqc25s
procedure executes a 25-point Clenshaw-Curtis integration and the integrand
is expected to be a function accepting a (Fortran) double precision value and
returning a double precision value. The equivalent C/C++ prototype for this
callback is \( \text{double(*fp)(double*)} \). However, C++ programming favors “call
by value” parameter passing and so the callable entity to be used as actual call-
back argument most likely has a function type \( \text{double(double)} \). This implies
that one needs to interpose a step in the evaluation that consists of dereferenc-
ing the pointer variable and then calling the actual target function. Our solution
provides this capability: using template specialization techniques, it compares
the signatures of the callable entity and of the callback and interposes such a
step when needed.

We propose a software solution that fulfills the needs described above and
discuss its use and implementation in the next section.

3. Overview of the software structure

The UML diagram in Figure 1 depicts the structure of the software solution
that we propose. An instance of the \texttt{Adaptor} class allows client code to obtain
a C or Fortran-compatible function pointer of type \( F_1 \) to a callable entity with
signature type \( F_2 \). The \texttt{Adaptor} uses a static \texttt{WrapperPool} that manages the
allocation of \texttt{Wrapper} structures used to wrap a callable entity. The imple-
mentation relies heavily on class templates, recursive template instantiation and
partial template specialization.

Different specializations of the \texttt{Wrapper} template are made depending on the
number of arguments in the call signature. A specialization for two-argument
signature types is shown in the diagram. The \texttt{Wrapper} class follows the approach
outlined in Listing 6, adding support for argument transformations. Multiple
instantiations of the \texttt{Wrapper} class template are made by the \texttt{WrapperMapFiller} class using recursive template instantiation.

The three significant classes in our software (\texttt{Adaptor}, \texttt{WrapperPool} and \texttt{Wrapper}) deal with providing a raw function pointer to the end user. We have systematically opted to flag abnormal behavior (e.g. failure to acquire a \texttt{Wrapper} from the \texttt{WrapperPool}) to the end user by returning a null pointer to the end user. The rationale for this approach is that this code can be used in a context where C++ is intermingled with C or Fortran in the call stack, possibly including third party binaries. Exception propagation across application binary interfaces and language boundaries is undefined and non-portable [11, item 62]. The null pointer approach does not have this drawback.

The solution is made available as a single header file consisting of some 850 lines of code, including extensive comments. The comments are formatted for extraction with doxygen [12] and thus provide documentation in HTML format. The code contains a significant amount of repetition of structure, a consequence of supporting call signatures with varying number of arguments. The distribution supports callable entities with up to nine parameters. A more compact header, where the repetitions are generated through macros is available, improving maintainability of the code. This header allows users to adapt callable entities with more than nine parameters, simply by redefining a single macro constant. A disadvantage however, is that the Boost [7] macro library is required for this header and this reduces usability and portability to a certain extent.

The following section discusses the interface and typical usage patterns of the \texttt{Adaptor} class, as well as the internals of the other classes depicted in Figure 1 and their implementation rationale.

4. Description of the individual software components

4.1. Adaptor

```cpp
template<typename F1, typename F2 = F1, unsigned int C = 5>
class Adaptor
{
private:
  Adaptor(Adaptor const&);
  Adaptor& operator=(Adaptor const&);
  typedef typename Implementation::CallSignature<F1, F2>::type BF2;
  typedef Implementation::WrapperPool<F1, BF2, C> WPool;
public:
  typedef F1 FunctionType;
  explicit Adaptor(function<BF2> f = function<BF2>() : WPool::adapt(f)) {}
  F1* get() const { return WPool::release(FPtr); }
  operator F1*() const { return WPool::release(FPtr); }
  Adaptor& bind(function<BF2> f) { WPool::bind(FPtr, f); return *this; }
  Adaptor& unbind() { WPool::unbind(FPtr, function<BF2>()); return *this; }
  bool hasTarget() const { return WPool::hasTarget(FPtr); }
  static unsigned int getCapacity() { return C; }
  static unsigned int getAvailable() { return WPool::getAvailable(); }
};
```

Listing 8: Code fragments of the \texttt{Adaptor} class.
Figure 1: UML diagram of the Adaptor software structure
The **Adaptor** class is the only client-facing class in the project. An instance of this class allows client code to obtain a raw function pointer to a callable entity. The **Adaptor** obtains this pointer by calling the **WrapperPool**’s **adapt** function. The **WrapperPool** manages the allocation of **Wrapper** classes that contain the raw function pointers and associated target objects. We will explain its interface using code fragment examples. More examples with extensive comments are included in the distribution of the software. In subsequent sections we will review the underlying implementation.

```cpp
typedef Adaptor<double(double), 10> A;
assert( A::getAvailable() == 10 );
Pow p3(3.0);
A a(p3);
A::FunctionType fp = a.get();
y = quad(0., 1., .01, A(p3).get());
assert( A::getAvailable == 9 );
...
```

Listing 9: Code fragment illustrating access to the adaptor and its free function pointer.

Listing 9 illustrates the basic use of the adaptor. The actual template parameters for the **Adaptor** template are the function type of the callback expected by **quad**, and the number of **Wrappers**. The raw function pointer is obtained by clients through the **Adaptor**’s **get()** method, its type through the **FunctionType** type definition. The number of available **Wrappers** is limited by a compile-time constant (the second template parameter). The methods **getCapacity()** and **getAvailable()** can be used to query the number **Wrappers** that were instantiated and the number that is currently available for use.

The lifecycle of the allocation of a **Wrapper** to a callable entity equals that of the **Adaptor** instance itself. Upon construction the **Adaptor** fetches a raw function pointer from the **WrapperPool**, upon destruction of the **Adaptor** instance this allocation is released.

```cpp
typedef Adaptor<double(double), 1> const A;
for ( int i = 0; i < 6; i++) {
    Pow p(i);
    A a(p);
    if ( a ) quad(0., 1., .01, a);
}
```

Listing 10: Code fragment illustrating the lifecycle and conversion of adaptor.

Listing 10 illustrates the lifecycle of an **Adaptor** instance. The instantiated **Adaptor** template is now associated with a **WrapperPool** that manages a single **Wrapper**. The raw function pointer of that **Wrapper** is reused in the iterations of the loop. In each iteration the **Wrapper**’s function pointer and target object are associated with a different functor. This does not raise an issue with respect to the limited size of the **WrapperPool**, as the destruction of **Adaptor** instance’s at the end of the iteration scope releases the allocation of the **Wrapper**.
The example also demonstrates the conversion of the adaptor to its raw function pointer in expressions where such pointer is expected. This increases usability by providing a shorthand e.g. in coding the test for successful adaptation signaled by a non-null raw pointer in the adaptor, or in coding the quad library call.

```cpp
typedef Adaptor<double(double), 1> const A;
A a;
try { a(1.0); } catch (bad_function_call& e) { cout << e.what() << endl; }
Pow p3(3.0);
Pow p4(4.0);
a.bind(p3);
A::FunctionType* fp = (a.bind(p3)).get();
quad(0., 1., .01, fp);
a.bind(p4);
quad(0., 1., .01, a);
...
a.unbind();
try { a(1.0); } catch (bad_function_call& e) { cout << e.what() << endl; }
```

Listing 11: Code fragment illustrating binding and unbinding of an adaptor.

In addition to binding the raw function pointer to a callable entity in the adaptor’s constructor, one can explicitly use the `bind` and `unbind` methods. This is illustrated in Listing 11. The default constructor instantiates an adaptor that acquires a raw function pointer, but whose target is an empty `std::function`. A call through the adaptor’s raw function pointer to the empty function will raise a `bad_function_call` exception (an exception type defined in the C++ Standard Library). The adaptor’s raw function pointer can be bound to a callable entity using the `bind` method. This method returns the `Adaptor` instance itself to allow expression chaining. A second `bind` call binds the raw function pointer to another callable entity, whereas a call to `unbind` breaks the binding (by binding the adaptor to an empty target). The `hasTarget` method reports whether the adaptor has already been bound to a non-empty callable entity.

The issue of exception above merits one remark. If an adaptor has been initialized with an empty function wrapper or has been subjected to the the `unbind` method, a call via its raw function pointer raises an exception. This potentially introduces problems (see Section 3) in mixed language environments. However, because we are dealing with a raw function pointer it is impossible to cloak it with a structure that would handle the situation.

### 4.2. WrapperMap and WrapperMapFiller

To generate the wrappers at compile-time, recursive template instantiation is used [13, ch17]. We introduce the `WrapperMap` class, see Listing 12, that is a Standard Library associative map with pairs of raw function pointers and pointers to callable entities. The elements are inserted into the map at compile-time by recursive instantiation of the `WrapperMapFiller` template.
As an example consider the WrapperMap constructor call in Listing 12, triggering the instantiation of the WrapperMapFiller<F1, F2, 5, 5> template. As a consequence, in the WrapperMapFiller constructor, the addresses of the static wrapper function Wrapper<F1, F2, 5, 5>::call and of the target object Wrapper<F1, F2, 5, 5>::fgTarget are inserted into the map. Subsequently, WrapperMapFiller<F1, F2, 5, 4> is instantiated. This recursion continues until the instantiation of WrapperMapFiller<F1, F2, 5, 0>. The explicit specialization provided for this case stops the recursion. The template parameter I in the definition of the Wrapper class controls the recursion and distinguishes each of the different Wrapper instantiations for a given set of values of F1, F2 and C.

```
template<typename F1, typename F2, unsigned int C, unsigned int I>
struct WrapperMapFiller
{
  WrapperMapFiller(map<F1, function<F2>>& wmap)
  {
    wmap[&Wrapper<F1, F2, C, I>::call] = &Wrapper<F1, F2, C, I>::fgTarget;
    WrapperMapFiller<F1, F2, C, I - 1> irrelevant_name(wmap);
  }
};

template<typename F1, typename F2, unsigned int C>
struct WrapperMapFiller<F1, F2, C, 0> { WrapperMapFiller(map<F1, function<F2>>&) {} };

template<typename F1, typename F2, unsigned int C>
struct WrapperMap : public map<F1, function<F2>>
{
  WrapperMap() { WrapperMapFiller<F1, F2, C, 0>(*this); }
};
...
WrapperMap<F1, F2, 5> m;
...
```

Listing 12: Code fragment illustrating the creation of adaptors.

The final result of this recursive instantiation process is a map filled with raw function pointers of type F1 and corresponding target objects of type function<F2>. In the following we refer to such a pair as a slot. These slots are the resources that will be managed the WrapperPool.

4.3. WrapperPool

A static WrapperMap instance is managed by the WrapperPool class that supports the acquisition and release of slots. As this class manages only static data, its design follows the Singleton pattern [14, Item 26]. The WrapperPool uses a separate “in use” map that maintains the slot’s availability status with a boolean for each slot. This map is generated using the same techniques discussed for the WrapperMap.

The WrapperPool::acquire member function scans the “in use” table for a slot that is not yet in use. Once such an entry has been found, the slot’s raw function pointer is returned. If no such entry is found, i.e. the whole table is in
use, a null pointer is returned. The WrapperPool::bind function associates a callable entity with the raw function pointer provided.

4.4. Wrapper

```cpp
template<typename R, typename S, typename P1, typename Q1, typename P2, typename Q2, unsigned int C, unsigned int I>
struct Wrapper<R(P1,P2), S(Q1,Q2), C, I>
{
  typedef CallSignature<R(P1,P2), S(Q1,Q2)> TT;
  static function<typename TT::type> fgTarget;
  static R call(P1 p1, P2 p2)
  {
    return fgTarget(TT::arg1::eval(p1), TT::arg2::eval(p2));
  }
};
```

Listing 13: Code fragment illustrating the Wrapper class for binary functions.

Listing 13 shows the definition of the Wrapper template specialization for binary functions. The class contains a static target object and a static function that forwards a function call to that object. In the forwarding process, we allow a number of argument transformations to map the function type of the wrapper’s static function to the target object’s function type. Our transformations extend the standard C++ argument conversion rules that apply when invoking the target object through the std::function wrapper. More specifically, they allow for arguments of pointer type to be dereferenced before they are passed to the target object.

The need to dereference function arguments occurs frequently when interfacing C++ code with C and Fortran libraries as they tend to accept function arguments by reference, even when semantically there is no need to do so. This is especially the case for Fortran libraries and many C libraries have followed this approach to attain compliance with Fortran calling conventions.

To implement the argument transformations, the CallSignature template is used that takes the function type of the wrapper function and that of the target object as template arguments. The CallSignature template determines for each function parameter whether a dereferencing operator should be applied to the argument before it is passed on to the target object. This choice is made at compile-time by specialization of the Pick template, for which a specialization is shown in Listing 14. Therefore, no additional conditional statements that could impact performance through branch prediction errors are introduced in the code path. The argument transformation is performed by invoking the static eval function on the CallSignature template for each function argument. This is done in the call method of the Wrapper class.

```cpp
template<typename T1, typename T2>
struct Pick<T1*, T2>
{
  static_assert(is_convertible<T1, T2>::value, Incompatible argument types);
  typedef T1 type;
  struct Arg { static T1 eval(T1* t1) { return *t1; } };
};
```

Listing 14: Code fragment illustrating the specialization of the Pick class that dereferences a function argument before it is passed to the target object.
The code in Listing 15 illustrates the use of argument transformation. It passes the exp function from Listing 1 with signature type `double (double)`, as a callback to the dqc25s function of the Fortran QUADPACK library with signature type `double (double*)`.

```c
void dqc25s_(double f(double), double a, double b, double bl, double br, ...);

Adaptor<double(double*), double(double)> const ad(exp);

dqc25s_(ad, &a, &b, &bl, &br, ...);
```

Listing 15: Code fragment illustrating automated call argument transformation for adapted Fortran functions (for conciseness only the first five arguments of the dqc25s_ function are shown).

5. Performance analysis

A function call made through a raw function pointer that is returned by the adaptor adds on the order of 30 machine instructions. These instructions take care of forwarding the call through the wrapper structure. Due to the static nature of these forwarding calls, the instruction count can be heavily optimized by the compiler.

In order to illustrate the performance impact in a typical use case, we have used the adaptor to enable the use of the Monte Carlo GSL integration routine `gsl_monte_plain_integrate` with a C++ functor, in order to compute the following 3-dimensional integrand:

\[
I = \int_{-\pi}^{+\pi} \frac{dk_x}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_y}{2\pi} \int_{-\pi}^{+\pi} \frac{dk_z}{2\pi} \frac{1}{(1 - \cos(k_x)\cos(k_y)\cos(k_z))} = \Gamma(1/4)^4 \frac{1}{4\pi^3}.
\]

This example integrand, which is related to the theory of random walks, is also used in the GSL reference manual and is taken from [15]. We use this example as it combines a routine that relies heavily on function call performance, with limited logic in the integration routine itself and in the function to be evaluated.

We measure the wallclock time taken to compute the integrand using a platform independent timing utility. On the test platform that is equipped with an Intel Q9550 quad core CPU, running SuSE Linux 12.1, the utility uses the `clock_gettime(CLOCK_REALTIME, ...) POSIX call. We vary the number of function calls \( f_n \) that the integration routine invokes for determining the integrand’s value. We take 100 independent samples per value of \( f_n \), by executing the test program multiple times using a bash script.

Table 1 summarizes the results for different values of \( f_n \), showing both the average and standard deviation on the time per call to the integration routine when using the adaptor, on the time per call when using an ad hoc wrapper approach as outlined in Listing 4, and on the relative difference between both.

As shown in the table, the use of our adapter comes at a low cost, with a relative increase in the execution time per call to the integration routine that
Table 1: Performance analysis results.

<table>
<thead>
<tr>
<th>$f_n$</th>
<th>$Adaptor_{rt}$ (ms)</th>
<th>$AdHoc_{rt}$ (ms)</th>
<th>$\frac{Adaptor_{rt}}{AdHoc_{rt}}$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0e3</td>
<td>$\mu$ 0.20</td>
<td>0.19</td>
<td>7.27</td>
</tr>
<tr>
<td></td>
<td>$\sigma$ 0.01</td>
<td>0.07</td>
<td>36.50</td>
</tr>
<tr>
<td>1.0e6</td>
<td>$\mu$ 166.38</td>
<td>165.57</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>$\sigma$ 3.04</td>
<td>4.77</td>
<td>2.73</td>
</tr>
<tr>
<td>1.0e9</td>
<td>$\mu$ 165481.10</td>
<td>164853.40</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>$\sigma$ 2538.64</td>
<td>1520.86</td>
<td>0.92</td>
</tr>
</tbody>
</table>

lies below 1% in the case of $f_n = 1.0e6$ and $f_n = 1.0e9$. For $f_n = 1.0e3$, high variance in the results can be observed. This is due to the very low runtime of the program. Timing a loop that executes the routine under test 1000 times instead of timing a single execution, leads to an average relative increase in runtime of 0.61% with a standard deviation of 0.47% which is in line with the other test results.

6. Installation instructions

6.1. Distribution and compilation

The program consists of a single C++ header file adapt2rfp.h, to be included by client code. All compilers that support `std::function`\(^1\) are able to compile the header’s code. On some compilers this requires a flag to enable C++0x or TR1 extensions (e.g. `-std=c++0x` or `-std=c++11` for g++). An overview of C++11 compliance for various new C++11 elements in the language and the Standard Library can be found at [16].

In practice this leads to the following version constraints on some of the popular C++ compilers:

- g++: Version 4.3 or later required
- Visual Studio: Version 2008 (with Visual C++ 2008 Feature Pack) or later required

We built and tested the software only on these two platforms. Other compilers such as Intel’s C++ or the Clang frontend for the LLVM compiler infrastructure and their libraries also provide `std::function`. In fact, most compilers and Standard Library combinations have provided this for some time now.

The test and example code is built using make. The Makefile in the src/ directory defines three macros:

- **CC**: name of the C compiler you want to use e.g. gcc

\(^1\) `std::function` is part of the new C++11 ISO standard
CXX : name of the C++ compiler you want to use e.g. g++

FC : name of the Fortran compiler you want to use e.g. gfortran

The first two are indispensable. If you do not have a Fortran compiler, leave the FC name blank (but do not delete the line with its definition!) and the make procedure will skip the example involving a Fortran library. To compile the test and example code, simply execute make all in the src/ directory. See also the top-level Readme.txt file.

6.2. Testing

The distribution includes a set of unit tests developed using Google’s C++ Testing Framework and a set of examples that extensively demonstrate the Adaptor’s use but also serves as additional tests. After compilation (see Section 6.1) one launches the tests and examples with the command make test in the directory src/. This executes all of the tests and all of the example programs.

The unit test sources can be found in the src/gtests subdirectory of the project and include the required source files for the Google testing framework. You can invoke the tests also from here with make test or by explicitly executing the gtestdriver binary constructed by the build step. For details on the test that are run see Section 7.

The example code for the Adaptor’s usage is located in src/demo_c/ for interaction with C and in src/demo_f/ for interaction with Fortran. The latter only applies if a Fortran compiler is installed on your system (see Section 6.1).

7. Test run description

The unit tests in the src/gtests subdirectory (see Section 6.2) can be executed by launching the gtestdriver executable. Results are provided in the format familiar for Google’s C++ Testing Framework, listing each test by name and test status (OK or FAILED). The unit test are organized in three test cases.

The test case AdaptorTest consists of 17 tests. It deals directly with the class Adaptor and its member functions: construction and destruction of an adaptor object, binding of function, functor, const-qualified functor, member function, const-qualified member function and binding of an empty function object. The test case also checks the functioning of polymorphism through the pointer produced by the adaptor.

The test case PoolTest consists of 23 tests and verifies the functioning of the WrapperPool datastructure. It checks capacity, acquisition and release of wrappers, ownership of raw pointer, handling of wrapper status (in use, bound, empty) and so on.

The test case SignatureTest consists of 10 tests and checks explicit instantiation of the templates for up to nine arguments in the function that is to be adapted, and tests use of the raw pointer produced by the adaptor.

The entire suite is comprised of 50 tests and provides a reasonably comprehensive coverage of the software.
The unit tests are supplemented by demo programs consisting of small functions that each provide example code for a particular use or application of the Adaptor. The demo program for interaction with C code consists of 27 examples, the program for interaction with Fortran code consists of 10 examples (so as not to have too much code duplication). For the latter, supplying a raw pointer both to Fortran functions and to Fortran subroutines in the library is demonstrated.

The first group of examples focuses on the syntactic transformation of the various callable types to pointer to function type. A second group focuses on the conversions and transformations that are interposed between the signature of the callable entity and the raw function pointer signature. The third group focuses on the reuse and lifetime of adaptors and associated raw pointers.

References


