A General Approach to Creating Fortran Interface for C++ Application Libraries

Yang Wang[1], Raghurama Reddy[1], Roberto Gomez[1], Junwoo Lim[1], Sergiu Sanielevici[1], Jaideep Ray[2], James Sutherland[2], and Jackie Chen[2]

[1] Pittsburgh Supercomputing Center, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A. ywg@psc.edu
[2] Sandia National Laboratories, Livermore, CA 94550-0969, USA

Summary. Incorporating various specialty libraries in different programming languages (FORTRAN and C/C++) with the main body of the source code remains a major challenge for developing scientific and engineering application software packages. The main difficulty originates from the fact that Fortran 90/95 pointers and C/C++ pointers are structurally different. In this paper, we present a technique that allows us to circumvent this difficulty without using any nonstandard features of these programming languages. This technique has helped us to develop a FORTRAN 90/95 interface for the GrACE library, which is written in C++ for facilitating spacial grid generation, adaptive mesh refinement, and load balance maintenance. The method outlined in this presentation provides a general guideline for the creation of the FORTRAN 90/95 interface for a C/C++ library. We show that this method is system independent and has low overhead cost, and can be easily extended to situations where building the C/C++ interface for a FORTRAN 90/95 application library is required.
Key words: FORTRAN, C/C++, pointers, inter-language calling, interface

1 Introduction

For maximal portability, the best approach to scientific and engineering application software development is to use a single programming language. In many cases, it is, however, undesirable or even unfeasible to do so, mainly because of the fact that many widely used numerical or graphical software libraries are not developed in a single programming language. Consequently, incorporating various specialty libraries in different programming languages (FORTRAN and C/C++) with the main body of the source code usually becomes a major headache in the process of application software development. The primary reason for this difficulty is the differences in the conventions for argument passing and data representation. A Fortran 90/95 pointer is quite different from a pointer in C/C++. While the argument passing conventions for the basic data types has been worked out fairly well, there are significant

difficulties in handling pointer data types as arguments. The primary reason for passing pointer information between C/C++ and FORTRAN is to pass references to dynamically allocated memory. In addition, the referenced memory may contain data values which are, in effect, also being exchanged and which we must therefore be able to reference from both languages. To our knowledge, no effective method is publicly known to allow FORTRAN 90/95 pointers to directly access the data space allocated in C/C++ routines. While some efforts such as Babel[DEKL04] are attempting to solve this problem in a more general purpose way, these methods are usually system dependent and involve significant overhead and learning curve.

In our recent effort to build a Terascale 3D Direct Numerical Simulation package for flow-combustion interactions, we developed a technique for building a FORTRAN interface for C++ libraries. This allows shielding the C++ implementation of the functionalities from our FORTRAN legacy code. Most importantly, the interface provides the access functions for FORTRAN 90/95 pointers, which can be one- or multi-dimensional, to access the data allocated in the C++ library. In our approach to the FORTRAN interface design, only the standard features of the programming language are used. The method outlined in this presentation provides a general guideline for the creation of the FORTRAN interface.

2 Inter-language Calling: Conventions and Challenges

It has become a widely-supported convention that the FORTRAN compilers convert the name of the FORTRAN subroutines, functions, and common blocks to lower case letters and extend the name with an underscore in the object file. Therefore, in order for C/C++ to call a FORTRAN routine, the C/C++ routine needs to declare and call the FORTRAN routine name in low case letters extended with an underscore. On the other hand, declaring the name of a C routine in lower case letters extended with an underscore makes the C routine name (without the underscore) callable from FORTRAN. Comparing FORTRAN and C/C++, the basic data types are fairly similar, e.g., `REAL*8` v.s. `double` and `INTEGER` v.s. `long`, although the exact equivalence between these data types depends on the computer system and the compilers. Besides the similarity between their basic data types, a FORTRAN subroutine is equivalent to a C function with a return type of `void`, while a FORTRAN function is equivalent to a C function with a non-void return type. However, there is a major difference between FORTRAN and C in the way they handle the arguments when making routine or function calls. FORTRAN in most cases passes the arguments by reference, while C passes the arguments by value. For a variable name in the argument list of a routine call from FORTRAN, the corresponding C routine receives a pointer to that variable. When calling a FORTRAN routine, the C routine must explicitly pass addresses (pointers) in the argument list. Another noticeable difference

between FORTRAN and C in handling routine calls is that FORTRAN routines supply implicitly one additional argument, the string length, for each `CHARACTER*N` argument in the argument list. Most FORTRAN compilers require that those extra arguments are all placed after the explicit arguments, in the order of the character strings appearing in the argument list. When a FORTRAN function returns a character string, the address of the space to receive the result is passed as the first implicit argument to the function, and the length of the result space is passed as the second implicit argument, preceding all explicit arguments. Whenever the FORTRAN routine involves character strings in its argument list and/or returns a character string as a function, the corresponding C routine needs to declare those extra arguments explicitly. Finally, we note that the multi-dimensional arrays in FORTRAN are stored in a column-major order, whereas in C they are stored in a row-major order.

The FORTRAN/C inter-language calling conventions summarized above have proved to be very useful in the software development for scientific and engineering applications that often requires incorporating various specialty libraries in different programming languages (FORTRAN or C/C++) with the main body of the source code. Despite the fact that the variables of basic data types passed between FORTRAN and C/C++ routines can be handled fairly well by the FORTRAN/C inter-language calling convention, one major difficulty remains to be resolved. That is we are not allowed to let a FORTRAN 90/95 pointer be passed from or to a C/C++ routine to make it associated with a C/C++ pointer, even though the pointers are of a basic data type. Unfortunately, this problem appears in many situations, and can become a major obstacle in the application software development. To give a better description of the problem, we discuss the example code shown in Table 2 which represents a fairly typical situation in reality when mixing FORTRAN code and C++ code. In this example, the FORTRAN subroutine `Application()` is a legacy code that needs to call `getGrid()` function to associate the three dimensional FORTRAN 90/95 pointer `grid` with the grid array created by a utility library, `MeshModule`, a FORTRAN 90/95 module specially designed for 3D mesh generation. The new utility library we intend to use is written in C++, and it contains a `Mesh` class that implements the functionality for 3D mesh generation and a function, `getGrid()`, for returning a C/C++ pointer associated with the memory space allocated for the grid. Unfortunately, we are required not to make any changes to the legacy FORTRAN code nor to the library code. (In many cases, the library source code is not even available.) The technical difficulty is that we are not allowed to let the FORTRAN code call the member function of `Mesh` class directly, since the FORTRAN code has no way to recognize the instance of `Mesh` class. Furthermore, even if the utility library provides a C routine `getGrid()` for the FORTRAN code to call, associating a FORTRAN 90/95 pointer directly with the C/C++ pointer returned by `getGrid()` is still not allowed, needless to mention one more difficulty that the FORTRAN 90/95 pointer here is a three dimensional array whereas the C/C++ pointer is represented by a one dimensional array.

Table 1. A problem for using FORTRAN 90/95 pointer to access the "`grid`" array allocated in the C++ code

FORTRAN 90/95: Application.f90	C++: Mesh.h

`SUBROUTINE Application()`
`  USE MeshModule, ONLY : &`
`  initMesh, getGrid, deleteMesh`
`  INTEGER :: n1, n2, n3`
`  REAL*8 :: geometry(3,3)`
`  REAL*8, pointer :: grid(:,:,:)`
`  ... ...`
`  READ(*,*)n1, n2, n3, &`
`  geometry(1:3,1:3)`
`  CALL initMesh(n1, n2, n3, &`
`  geometry)`
`  grid => getGrid()`
`  ... ...`
`  CALL deleteMesh()`
`END SUBROUTINE Application`
	`class Mesh {`
	`public:`
	`  void Mesh(long *,`
	`  long *, long *,`
	`  double *);`
	`  void ~Mesh();`
	`  double *getGrid() const;`
	`  long getSizeX() const;`
	`  long getSizeY() const;`
	`  long getSizeZ() const;`
	`private:`
	`  double *grid;`
	`  long nx, ny, nz;`
	`  double geometry[9];`
	`}`

Generally speaking, when making a C++ numerical library usable by a FORTRAN application, we usually have to resolve these following problems: how to construct and destruct a C++ object via FORTRAN routine calls, how to access the public member functions of a C++ object via FORTRAN routine calls, how to make a multi-dimensional FORTRAN 90/95 pointer an alias of a one dimensional array allocated dynamically in the C/C++ routine, and finally how to make a memory space allocated dynamically in C/C++ routine accessible by a FORTRAN 90/95 pointer.

There are some possible ways to get around these problems. For example, we can build a simple main driver in C/C++ that calls the constructor and the destructor of the classes defined in the library. The drawback of this approach is that the life of the objects spans over the entire job process. Alternatively, we can "wrap" the C++ member function, e.g., `getGrid()`, with a C-style function and call the constructor and the destructor, e.g., `Mesh(...)` and `~Mesh()` in the "wrapper". But if the member function needs to be called frequently, this approach can be very costly. As far as the FORTRAN 90/95 and the C/C++ pointer association is concerned, the CNF library[CNF] provides one of possible solutions. In the CNF library, a set of C macros are defined to help handling the difference between FORTRAN and C character strings, logical values, and pointers. For pointers, in particular, CNF does not allow FORTRAN 90/95 pointers to be associated with C pointers directly, rather, it uses FORTRAN `INTEGER` type to declare the C pointer passed into the FORTRAN routine so that the corresponding `INTEGER` variable represents the address of the array. Unfortunately, this scheme of using FORTRAN `INTEGER`s to hold pointer values only works cleanly if the length of an `INTEGER` is the

same as the length of the C generic pointer type `void*`. In order to use CNF, it requires to add macros to the C++ library or to implement a C "wrapper" for the library with the macros built in. In either case, the readability of the code becomes problematic. Another approach is to use CRAY or Compaq pointers, which are nonstandard FORTRAN pointers allowed to be associated with C/C++ pointers directly. Unfortunately, only a few FORTRAN compilers support these types of pointers. Finally, we mention the Babel[DEKL04] project, that attempts to solve the problem of mixing languages in a more general way. So far, Babel only works on Linux systems, and its support for FORTRAN 90/95 pointers is still questionable.

3 Approach to the Solution

Our approach is to build a "wrapper" for the C++ library that hides the implementation details of the C++ library from the FORTRAN application. The "wrapper" handles the request from FORTRAN calls to create and destroy objects defined in the C++ library, and provides functions to return FORTRAN pointers aliased to the memory space allocated in the C++ library. It plays an intermediate role between the FORTRAN application and the C++ library and helps to minimize the changes that need to be made to the application source code.

The architecture of the "wrapper" is sketched in figure 1. The "wrapper" is made of two layers: a C-Layer and a FORTRAN-Layer, and both are written in "standard" C/C++ and FORTRAN 90/95 programming languages, together with the inter-language calling conventions summarized in the previous section. The C-Layer contains, in its global scope, a pointer to the C++ object that needs to be created or destroyed. In addition to providing a C-type interface between the FORTRAN-Layer and the public functions defined in the C++ library, the C-Layer calls the "bridge" functions, which are a set of FORTRAN routines provided by the FORTRAN-Layer, to make the FORTRAN 90/95 pointers aliased to the memory allocated dynamically in the C++ library. The FORTRAN-Layer is composed of a FORTRAN 90/95 module and the "bridge" functions, which are external to the module. The FORTRAN 90/95 module provides a set of public functions for the FORTRAN application routines to call. Each of these public functions corresponds to a public function implemented in the C++ library, and it calls the corresponding C function implemented in the C-Layer that in turn calls the corresponding public function in the C++ library. The FORTRAN 90/95 module also provides an alias function with `public` attribute for each "bridge" function to call. In its global space, the FORTRAN 90/95 module holds a set of one- or multidimensional FORTRAN 90/95 pointers that need to be aliased to the memory space allocated in the C++ library.

Aliasing a FORTRAN 90/95 pointer to a memory space allocated in the C++ library takes the following steps. Upon an access function call from the

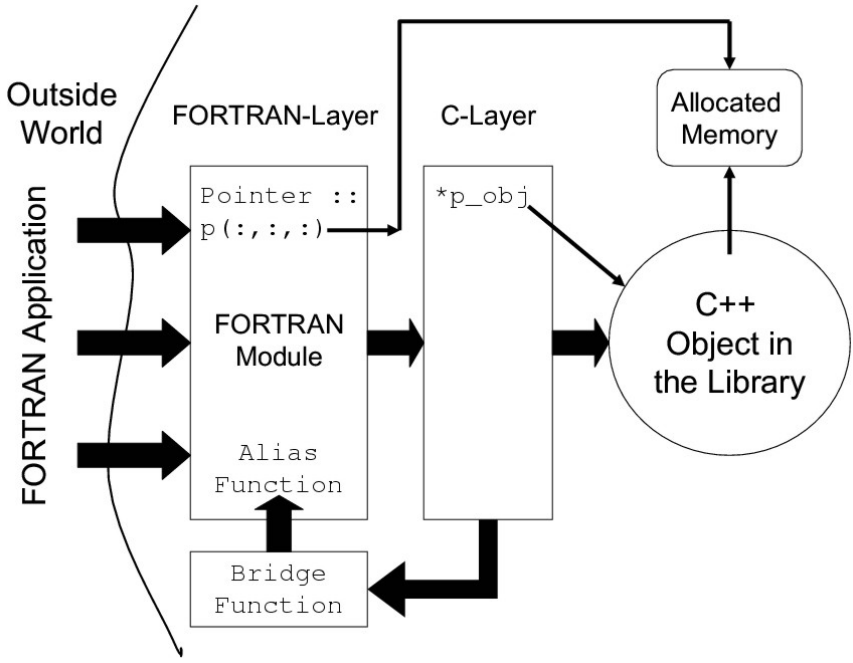

Fig. 1. The architecture of the "wrapper", a FORTRAN interface, for the C++ library. The "wrapper" is made of a C-Layer and a FORTRAN-Layer.

FORTRAN application that asks for the return of a FORTRAN 90/95 pointer, the FORTRAN 90/95 module in the FORTRAN-Layer calls the corresponding C routine in the C-Layer. The C routine calls the corresponding access function in C++ library to get the C/C++ pointer pointing to the memory space and then calls the "bridge" function to pass the C/C++ pointer into FORTRAN. The "bridge" function gets the C/C++ pointer together with its size from the caller and declares it as a FORTRAN array. The "bridge" function then calls the corresponding alias function in the FORTRAN 90/95 module to pass in the array. The alias function declares the array with appropriate dimension together with `TARGET` attribute, and then assigns the array to the FORTRAN 90/95 pointer using the operator "`=>`". Finally, the access function in the FORTRAN 90/95 module returns the FORTRAN 90/95 pointer to its caller (the FORTRAN application).

We again use the example in Table 2 to illustrate the method described above. The "wrapper" for `Mesh.cc`, the C++ library, is made of a FORTRAN-Layer that contains `MeshModule.f90` (see Table 3) and `grid_Bridge.f90` (see Table 4) and a C-Layer that contains `MeshWrapper.cc` (see Table 5). The private member variable `grid(:,:,:)` in the FORTRAN 90/95 module `MeshModule` is a pointer to be associated with the grid space allocated

Table 2. An example of the FORTRAN 90/95 module in the FORTRAN-Layer

MeshModule.f90

```
MODULE MeshModule
PUBLIC : initMesh, getGrid, deleteMesh, aliasGrid
PRIVATE
   INTEGER :: nx, ny, nz
   REAL*8, pointer :: grid(:,:,:)
CONTAINS

SUBROUTINE initMesh(L,M,N,g)
   INTEGER :: L,M,N
   REAL*8 :: g(9)
   CALL c_initMesh(L,M,N,g)
END SUBROUTINE initMesh
!============================
SUBROUTINE deleteMesh()
   CALL c_deleteMesh()
END SUBROUTINE deleteMesh
!============================
FUNCTION getGrid() result(p)
   REAL*8, POINTER :: p(:,:,:)
   CALL c_getGrid()
   p => grid(1:nx,1:ny,1:nz)
END FUNCTION getGrid

!============================
SUBROUTINE aliasGrid(n1,n2, &
                     n3,a)
   INTEGER :: n1,n2,n3
   REAL*8, TARGET ::        &
                  a(n1,n2,n3)
   nx = n1
   ny = n2
   ny = n3

   grid => a(1:nx,1:ny,1:nz)

END SUBROUTINE aliasGrid
!============================

END MODULE MeshModule
```

Table 3. An example of the "bridge" function in the FORTRAN-Layer

grid_Bridge.f90

```
SUBROUTINE grid_Bridge(nx, ny, nz, a)
   USE MeshModule, only: aliasGrid
   INTEGER, INTENT(IN) :: nx,ny,nz
   REAL*8, INTENT(IN) :: a(nx*ny*nz)
   CALL aliasGrid(nx, ny, nz, a)
END SUBROUTINE grid_Bridge
```

by `Mesh.cc`. The member function `initMesh` in the module calls routine `c_initmesh_` in `MeshWrapper.cc` to create a `Mesh` object; function `deleteMesh` calls routine `c_deletemesh_` to destroy the `Mesh` object; function `getGrid` calls routine `c_getgrid_` and returns a pointer, aliased to `grid(:,:,:)`, to the caller; function `aliasGrid`, called by the bridge function `grid_Bridge`, makes the pointer `grid(:,:,:)` aliased to the grid array originated as a C/C++ pointer `p_mem` from routine `c_getgrid_` in `MeshWrapper.cc`. The pointer `*p_obj` in the global space of `MeshWrapper.cc`

is created as an instance of class `Mesh` after routine `c_initmesh_` is called, and

Table 4. An example of the C-Layer

MeshWrapper.cc

```
#include "Mesh.h"
Mesh *p_obj;
extern void grid_bridge_(long *, long *, long *, double *);
extern "C" {
void c_initmesh_(long *nx, long *ny, long *nz, double *geom) {
   p_obj = new Mesh(nx,ny,nz,geom);
}
void c_deletemesh_() { delete p_obj; }
void c_getgrid_() {
   double *p_mem = p_obj->getGrid();
   long nx = p_obj->getSizeX();
   long ny = p_obj->getSizeY();
   long nz = p_obj->getSizeZ();
   grid_bridge_(&nx, &ny, &nz, p_mem);    p_mem = 0;
}
}
```

is deleted after routine `c_deletemesh_` is called. The routine `c_getgrid_` calls the member function `getGrid` of the `Mesh` object, to obtain the address of the grid space and store the address in pointer `*p_mem`. After calling `grid_bridge_`, `c_getgrid_` passes the address together with the size of the grid space in all dimensions, `nx`, `ny`, and `nz`, into the FORTRAN routine `grid_Bridge`, in which the grid space is declared as a FORTRAN array `a(nx*ny*nz)`. This array together with its size in each dimension is then passed into `aliasGrid`, where it is declared as a three dimensional array with a `TARGET` attribute so that the FORTRAN 90/95 pointer `grid(:.:.:)` is allowed to be aliased to it.

Obviously, the method outlined above is system independent, since the only nonstandard features of the programming languages used here are the widely supported inter-language conventions, summarized in Section 2. The "wrapper" does not involve any heavy numerical computations and is only used for passing in and out the parameters by reference without making copy of sizeable data, and therefore it has very low overhead cost. The "wrapper" for the C++ library is able to hide all the implementation details from the FORTRAN application, and makes the C++ library behave as a FORTRAN library with interfaces provided by the FORTRAN 90/95 module in the "wrapper". It becomes unnecessary for the FORTRAN application to be aware of the existence of the C++ objects and the C/C++ pointers behind the scene. In the example presented above, the FORTRAN 90/95 module in the "wrapper" has the same module name and public functions as the module

used in `Application()` (see Table 2) so that there is no need for us to make any changes to `Application.f90`. In practice, the situation is very unlikely to be as simple as this. We can choose to build an intermediate FORTRAN-Layer between the "wrapper" and the FORTRAN application and let this intermediate FORTRAN-Layer call the FORTRAN public functions provided by the "wrapper". In this way, the FORTRAN application does not require any changes to its source code.

4 Applications

The technique presented in the previous section has been applied to the development of a FORTRAN interface for GrACE, a C++ library facilitating mesh-management and load-balancing on parallel computers. Our legacy application code, namely S3D, is written in FORTRAN 90/95 and is used for solving the Navier-Stokes equations with detailed chemistry. The FORTRAN interface for GrACE made possible to incorporate the GrACE library into S3D. The new application package can now use the functionalities implemented in GrACE for grid generation and dynamic load balance maintenance. Our ultimate goal is to develop, based on the S3D code, a terascale 3D direct numerical simulation package, capable of performing integrations at multiple refinement levels using adaptive mesh refinement algorithm while maintaining load balance, for flow-combustion interactions.

The "wrapper" for a C++ library introduced in the previous section can be turned around and viewed differently. That is, from a different perspective, we can consider that the FORTRAN-Layer plus the C-Layer is a "wrapper" for the FORTRAN application and the C++ library/application interacts with the FORTRAN application via the C-Layer acting as an interface. Therefore, the same method can also be applied to build a C/C++ interface for a FORTRAN library/module, except that the bridge functions and the alias functions should now be written in C and should belong to the C-Layer. And most importantly, it makes possible to associate a C/C++ pointer with a FORTRAN 90/95 pointer array allocated in FORTRAN. Based on this observation, we have developed a systematic way of building CCA (Common Component Architecture) [AGKKMPS99] components for FORTRAN modules, subroutines, and/or functions without having to change the FORTRAN source code. The CCA components are the basic units of software that can be combined to form applications via ports which are the abstract interfaces of the components. Instances of components are created and managed within a CCA framework, which also provides basic services for component interoperability and communication. Briefly speaking, it takes two major steps to component-wize a FORTRAN module, subroutine or function. Firstly, we use the method presented in the previous section to build a C/C++ interface for the FORTRAN code, and secondly, we follow the standard procedure to

componentize the resulting C/C++ interface. A detailed description of this work will be presented in a future publication.

5 Conclusions

In summary, we have developed a general method for building a FORTRAN interface for a C++ library. This same method can also be applied to building a C/C++ interface for a FORTRAN library. The interface is made of a FORTRAN-Layer and a C-Layer. The FORTRAN 90/95 module in the FORTRAN-Layer provides an interface for the FORTRAN application and the C-Layer provides an interface for the C/C++ application. The "crux" of this method for associating a FORTRAN 90/95 pointer with a C/C++ pointer (or vise versa) is the bridge function that converts a C/C++ pointer into a FORTRAN array (or vice versa) and the alias function that makes a FORTRAN 90/95 pointer array alias to the FORTRAN array associated with the C/C++ pointer (or assigns a C/C++ pointer the address value of the FORTRAN array.)

This approach has several advantages. It is portable and has very low overhead cost. The requirement for making changes to the legacy application code is minimal and can actually be avoided. We have applied this method to build a FORTRAN interface for the GrACE library in our SciDAC project, and we have also used the same technique to componentize some of the FORTRAN 90/95 modules in our FORTRAN applications.

Acknowledgement. This work is supported in part by U.S. Department of Energy, under SciDAC Scientific Application Pilot Programs (SAPP), Contract No. DE-FC02-01ER25512.

References

[AGKKMPS99] Armstrong, B., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and Smolinski, B.: Toward a Common Component Architecture for High-Performance Scientific Computing. *8th IEEE International Symposium on High-Performance Distributed Computing*, Redondo Beach, CA (1999)

[CNF] www.starlink.rl.ac.uk/static_www/soft_further_CNF.html

[DEKL04] Dahlgren, T., Epperly, T., Kumfert, G., and Leek, J.: Babel Users' Guide. www.llnl.gov/CASC/components/docs/users_guide.pdf (2004)

[PB00] Parashar, M. and Browne, J.C.: Systems Engineering for High Performance Computing Software- *The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement*. IMA Volume 117, Editors: S.B. Baden, N.P. Chrisochoides, D.B. Gannon, and M.L. Norman, Springer-Verlag, pp. 1-18 (2000)