CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability

C.E. Rasmussen
Advanced Computing Laboratory
Los Alamos National Laboratory

K.A. Lindlan
Department of Computer Science
University of Oregon

B. Mohr, J. Striegnitz
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich

Abstract

The relative simplicity and design of the Fortran 77 language allowed for reasonable interoperability with C and C++. Fortran 90, on the other hand, introduces several new and complex features to the language that severely degrade the ability of a mixed Fortran and C++ development environment. Major new items added to Fortran are user-defined types, pointers, and several new array features. Each of these items introduce difficulties because the Fortran 90 procedure calling convention was not designed with interoperability as an important design goal. For example, Fortran 90 arrays are passed by array descriptor, which is not specified by the language and therefore depends on a particular compiler implementation. This paper describes a set of software tools that parses Fortran 90 source code and produces mediating interface functions which allow access to Fortran 90 libraries from C++.

1.0 Introduction

Fortran is usually an integral part of the computing environment at a scientific institution like Los Alamos National Laboratory (LANL). At LANL, there are many legacy Fortran applications and libraries, as well as scientific programmers who use Fortran as a language of choice. Legacy Fortran applications have frequently been written in Fortran 77 using static arrays as the dominant data structure, e.g. [1]. These static arrays were well suited for the time as they easily mapped onto the fixed grid models of the legacy codes.

However, as scientific models have increased in complexity, it has been found that Fortran 77 is not expressive enough to provide the language support that many of the new models require. For instance, the trend has increasingly been for computational fluid dynamic (CFD) models to require high resolution in certain regions of interest (to resolve the physics), but in other modeled regions, physical parameters may vary slowly enough that a courser grid is appropriate, e.g. [2]. Thus, the tendency has been for scientific

models to move from static fixed grids to particle and fluid meshes that adapt to flow and other dynamic, model parameters.

Fortran 90 provides many new features that are attractive to Fortran 77 programmers [3]. The most important of these include data structures and user-defined types, many new array features and operations, pointers, increased support for code modularization, and improved type safety. The new array features include the ability to process entire arrays as a single object and to dynamically allocate (and deallocate) arrays. While the rank (i.e., the number of dimensions) of an array must be specified at compile time, the shape (i.e., upper and lower bounds in each dimension) may be deferred until runtime. These new features in Fortran 90 allow the increased complexity of the newer scientific models to be more easily and naturally expressed.

At the same time as scientific programmers have been migrating to Fortran 90, many have also begun to use C++. In addition, younger scientific programmers frequently use C++ because of the dominance of C and C++ in the educational and commercial environments. This has led to a mixed programming environment of Fortran 90 and C++. While Fortran 77 and C are largely able to coexist in a mixed programming environment (for example, libraries such as MPI frequently have both a Fortran and a C interface [4, 5]), this is not the case with Fortran 90 and C++. One of the main problems arises when one wishes to share arrays and user-defined types between Fortran 90 and C++ modules.

Arrays and pointers to an array, for example, are passed via a descriptor to explicit procedure interfaces. Unfortunately, there is no Fortran standard that describes the layout of an array descriptor. Thus, while a pointer to the memory address of a Fortran 90 array may be passed to C++ (via an implicit interface), there is no standard way for C++ to call a Fortran 90 procedure that expects an array or a Fortran pointer to an array. In essence, there is no way to cast a memory address to an array in Fortran. As stated by Adams et al. [3], "currently, the only way to guarantee consistent interfaces across implementations is to write all procedures in standard Fortran."

This paper addresses this situation by describing CHASM, a set of software tools and methods that use static code analysis to generate adapter functions that bridge the divide between C++ and Fortran 90 in a language-conforming way. The general technique is to first parse Fortran source code to discover a list of procedures and their interfaces, including the number of arguments and their types. The Program Database Toolkit (PDT) [6] is used to accomplish this task. Then for each procedure in this list, an adapter procedure is automatically generated that allows C++ to call the desired Fortran procedure. Within this adapter function, argument type conversion is performed if necessary and the associated Fortran 90 procedure is called (with return values also converted as necessary).

The primary focus of this paper is to describe the adapter functions that are used as a bridge in CHASM to call Fortran 90 procedures from C++. The emphasis will be on the passing of arrays as parameters to Fortran procedures. Similar techniques could be used to generate adapter functions to allow Fortran 90 to call C++ or to share user-defined

types between the two languages, but this is outside the focus of this paper and will be addressed later.

The paper proceeds as follows. First, the PDT and associated tools are described that allow for the static analysis of Fortran 90 source code. Then, the general design of the adapter functions is covered, with particular attention given to the passing of array type parameters. Finally, a C++ array class is described, that is used to wrap Fortran 90 arrays so they can be shared with the C++ environment.

2.0 Program Database Toolkit

The Program Database Toolkit (PDT) [6] provides a framework in which application developers can access programming language constructs and modify and generate object-oriented software. As such, it is an extremely useful tool and can be used in a variety of different settings.

Two existing libraries that make use of the PDT are TAU (Tuning and Analysis Utilities) [7] and SILOON (Scripting Interface Languages for Object-Oriented Numerics) [8]. TAU uses the PDT to gather information about the location of function entry and exit points, and uses this information to insert calls to profiling and tracing routines so that existing source code can be automatically instrumented for performance behavior. SILOON automatically generates bridging code to provide users with the ability to program in a high-level scripting language (such as Python and Perl) while accessing existing C++ libraries. SILOON uses the PDT to gather information about methods and functions in C++ source code (including the number and type of function parameters), and uses this information to generate bridging code so that these functions can be called from a scripting language.

The PDT framework works by first parsing source code (utilizing commercial compiler front ends) and then processing the resulting intermediate language (IL) trees. As it traverses an IL tree, a Fortran 90 (or C++) IL analyzer extracts information on high-level constructs and source code locations that is needed by analysis tools and code-generating applications. This information is then made available in a human-readable, "program database" (PDB) file. The DUCTAPE library [6] is then used to access the contents of the PDB file. The second release of PDT extends support to Fortran 90 and enhances the handling of C++ templates and template instantiations.

The IL analyzers process IL trees that are constructed during the parsing of source code. Based on the Mutek Fortran 90 [9] and Edison Design Group (EDG) C++ v2.45 [10] front ends, respectively, the Fortran 90 and C++ IL analyzers work similarly but operate on language-specific constructs. Both traverse IL trees, reporting information on designated, high-level constructs as they are encountered. Separate traversals for source files, routines, user-defined types, other types, and other entities allow restructuring of the reported information.

The output of the IL analyzers is a "program database" (PDB) file consisting of descriptions for various language entities. Each description identifies an item and lists its features. Information on source files, routines, Fortran 90 user-defined types and modules, C++ classes, structs and unions, other types, as well as templates, namespaces, and macros, is contained in a PDB file. A number of changes in the database format (from PDT version 1.0) were necessary to accommodate the addition of the Fortran 90 IL analyzer.

DUCTAPE is a C++ library that provides a common, object-oriented API to the PDB files produced by both IL analyzers. Each item type of the PDB format is represented by a class having a corresponding name (e.g., the pdbRoutine class describes routines). Information about the PDB items is accessible through member functions of the DUCTAPE classes. Common attributes are factored out into generic base classes. A single class hierarchy accommodates both Fortran 90 and C++. With the DUCTAPE library, PDT provides some useful static analysis tools. These include:

- `pdbconv` converts PDB files to a more readable format;
- `pdbhtml` creates web-based documentation;
- `pdbmerge` merges PDB files from separate compilations; and
- `pdbtree` displays file inclusion, class hierarchy, and call graph trees.

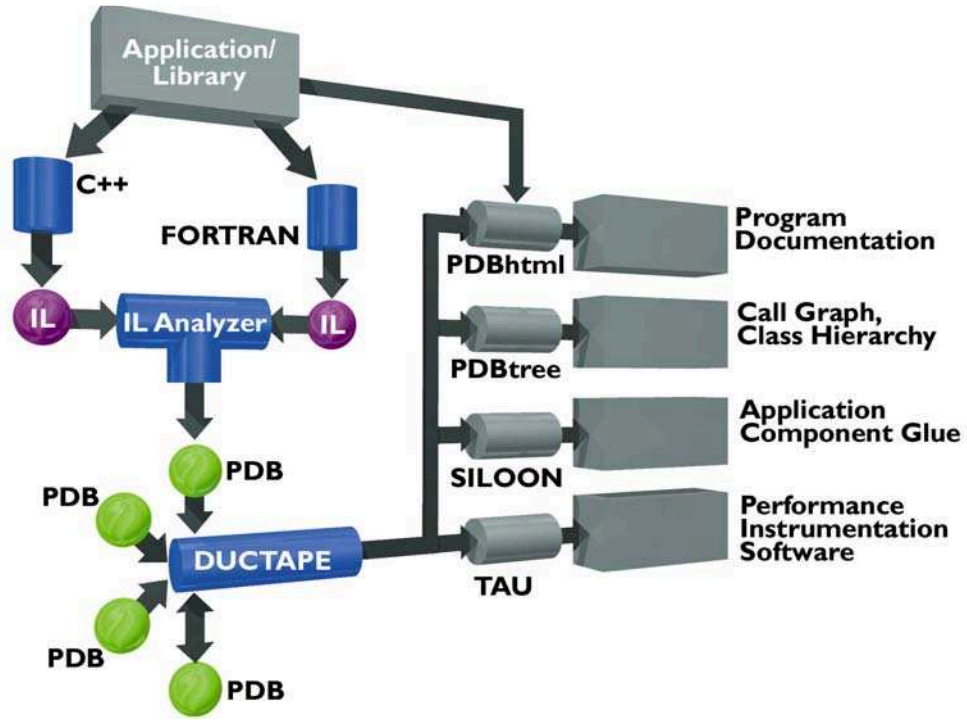

Figure 1. A schematic diagram of the static analysis of an application library by the PDT. Source code is parsed and the resulting intermediate language is processed and written to a PDB file that can be read using DUCTAPE, as shown in the left hand side of the figure. Examples of existing applications making use of the final output are shown in the right hand side of the figure.

3.0 General Design

Perhaps the easiest way to understand the design of the CHASM C++ to Fortran 90 interface is to compare it with a client/server relationship. Imagine that the C++ code is a client making calls on a Fortran 90 server. In traditional client/server software, there would be client-side stub and server-side skeleton code to provide an adaptive connection between differing client and server computer architectures and languages. These stub and skeleton interfaces must marshal function parameters in a way that accommodates existing differences between computer architectures and languages.

In a similar manner, the CHASM C++ to Fortran 90 interface procedures must accept client (C++) calls, marshall function parameters into Fortran 90 types, and, finally, to make the Fortran 90 procedure call. The CHASM generated software fulfills the role of an adapter pattern as described by Gamma et al. [11]. The call sequence is shown schematically in Figure 2.

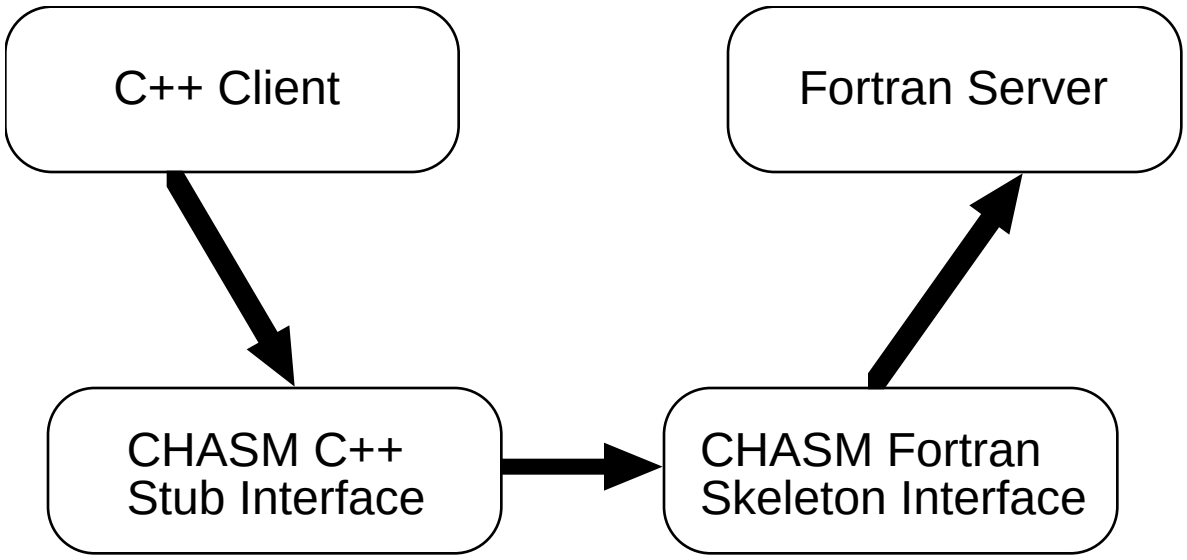

Figure 2. A schematic diagram showing the client/server analogy of the C++ to Fortran call sequence. The C++ client makes a call to a Fortran procedure through C++ stub and Fortran skeleton intermediaries. The C++ stub and Fortran skeleton interfaces marshal procedure parameters between C++ and Fortran.

A key distinction between traditional client/server code (e.g., the Common Object Request Broker Architecture (CORBA) [12, 13]) is that no interface definition language (IDL) is required to generate the adapter procedures. All information necessary to generate the adapters is discovered during the CHASM static analysis stage using tools from the PDT.

The client stub adapters are inline C++ functions that hide users from the different symbol-naming conventions used by Fortran compiler vendors. For instance, some compil-

ers require one or more leading underscores before symbol names, while others do not. Likewise, some Fortran compilers write symbols in all upper-case letters, while others do not. The CHASM convention is to use the procedure name as it is written in the Fortran code, but prepend the C++ F90 namespace. Suppose the original Fortran procedure is named Compute(), within a Fortran module named CFD. Then this procedure would be callable from C++ with the fully resolved name, F90::CFD::Compute().

The server skeleton procedures are written in Fortran 90. This allows the server calls to be made entirely within the Fortran language and in a compiler-independent fashion. The interfaces to these procedures are C-callable (implicit Fortran functions), and only standard C data types are passed in. This implies that special consideration must be given to Fortran 90 array data types in order to marshal to and from C++ and Fortran 90 as explained below. Standard data types (integers, floating point numbers, etc.) are easily handled and are not discussed further.

3.1 Arrays, Fortran Pointers and Dynamic Memory Allocation

As noted in the introduction, a major problem to overcome is that Fortran 90 arrays are passed by descriptor, which have no standard (compiler-independent) representation. To overcome this problem within the Fortran 90 language, several assumptions and design choices are made:

1. All allocation of array memory must be made from Fortran (at least for those arrays that are to be shared between Fortran and C++). This allows these arrays to be accessible in Fortran and prevents the compiler type system from complaining. Memory allocation within Fortran is necessary because Fortran 90 has no notion of type casting; it is simply not part of the language to be able to cast a memory address to a Fortran 90 array. To overcome this, a series of C-callable functions (based on array rank and element type) are provided for array memory allocation from C++. These functions are usually not called directly by users, who should access arrays through the provided C++ array class (described below). Array memory allocation is normally done within the constructor of the C++ array class. This class is a proxy (see [11]) of the Fortran 90 array.

2. A consequence of point 1 is that arrays are laid out in column major (as opposed to standard C, row major) ordering. Users are isolated from this by the C++ array proxy class, but should be aware of the memory layout for performance considerations.

3. A Fortran 90 array is represented by an integer handle in C++, maintained as an instance variable in the C++ array proxy class. The C++ stub interfaces automatically marshal an array reference to an integer handle when an array is required as a parameter to a Fortran 90 procedure.

4. Because Fortran arrays are represented by integers in C++ (point 3), there must be an association between the integer handle and the actual array in Fortran. This is accomplished via a set of global arrays (based on array rank and element type) in Fortran containing what are essentially pointers to the actual Fortran 90 arrays. The C++ integer

handle is an index into one of these arrays. There is an array for each dimension and array element type seen in the static analysis of the Fortran 90 code by the PDT. There must be separate arrays to meet the type requirements (no casting) of the Fortran language. What is stored in the arrays of handles is not a Fortran 90 pointer type, but a user-defined type wrapping a Fortran array pointer, as the language does not allow for arrays of pointers.

5. The arrays of pointers to arrays (discussed in point 4) are filled lazily whenever seen by an adapter interface. Suppose a Fortran procedure that is called from C++ returns a Fortran 90 array. The adapter code first checks if this array has been seen before (based on memory address, see point 6). If so, it simply returns the integer handle for this array. If not, it creates a pointer wrapper for the array, stores it in the proper array of handles (based on type and rank), and passes the index of the wrapper on to C++.

6. Fortran 90 must be coerced into giving up the memory address of a Fortran array. To maintain backward compatibility with Fortran 77, a Fortran array is passed by address to an implicit procedure. The memory address of a Fortran array obtained in this fashion is stored within the C++ array proxy for efficient memory access within C++. Note that Fortran array access from C++ is via an address into memory rather than a series of function calls for each individual element of the array.

7. A copy of what is essentially the Fortran array descriptor is stored in the C++ array proxy. This makes array meta information (rank, shape, and index ranges) accessible within C++. The interface for this information is given in the Appendix as the ArrayDesc class.

8. Each Fortran compiler has its own naming convention for procedure names written to object files and accessed by the linker. Normally, users must be directly aware of this convention, and call the proper symbol name from C when calling Fortran. Inline C++ stub interfaces are automatically generated to hide this nuisance from the user.

9. Fortran 90 pointers within C++ are treated the same as arrays. In other words, no distinction is made between pointers and arrays in the C++ environment.

3.2 Code Generation

There is a substantial amount of information unique to each Fortran 90 module that is analyzed by the PDT, and code that must be generated to allow for C++ calls of Fortran procedures. This information includes the number and types of arrays that are used in the Fortran 90 environment, specifically those that are passed in procedure calls. It also includes a list of procedure calls declared within the module and the parameter and return types for each procedure. Adapter stub and skeleton functions must be generated for each Fortran procedure to provide the CHASM bridge between C++ and Fortran. For each array rank and array type, code for a pointer wrapper must be generated and an array must be created to store these wrappers. Inline C++ code isolating the user from the Fortran compiler symbol-naming convention must also be generated.

Fortunately, the information required is readily available from the PDT once the Fortran code has been analyzed. Code generation is also easily accomplished. The steps of static analysis and code generation are similar to those already encountered in Figure 1.

3.3 Potential Problems

While the power of static code analysis allows for many of the traditional language interoperability issues to be overcome within the respective C++ and Fortran 90 languages, there still exist some potential problems.

A major problem is inherent in the management of memory assigned to each array. A C++ array proxy contains a pointer directly to Fortran array memory. Errors arise if this memory is deallocated in Fortran, leaving a dangling reference in C++. Unfortunately, nothing can be done to avoid this without direct user involvement. Before freeing an array in Fortran, a user should call F90ArrayDealloc(), a procedure provided to notify the C++ environment that an array is being freed and to nullify any potential proxies to this array.

As with any memory management scheme without direct garbage collection support, users should clearly document the usage of their code so that safe memory management is possible.

4.0 C++ Array Proxy Class

A C++ array class has been created to provide an object-oriented interface to Fortran 90 arrays. This is a proxy class since much of the functionality (e.g., memory storage) of the C++ class is actually provided by an associated Fortran 90 array. There are several issues that the array proxy class must handle: memory management, static typing properties, and array indexing. A discussion of these issues and their solution via the array proxy class is contained in this section.

There are two categories of Fortran functions that form the basic interface for memory management of arrays that are shared between C++ and Fortran. First, since all array memory is allocated by Fortran (to appease the strict type requirements of Fortran 90), there is a family of C-callable functions that is used to allocate arrays. These functions have names like F90MallocXY, where X is a placeholder for the type of the array element and Y is a placeholder for the rank of the array. For instance, F90MallocINT3 is used to allocate an integer array of rank 3. The result of a call to F90MallocXY is an instance of the class ArrayDesc (see the Appendix), which contains a pointer to the raw memory that is associated with the Fortran array, along with additional meta information describing the array. Similarly, the F90FreeXY family of functions free array memory when required.

These function families are generated as needed by CHASM, but are not called directly by users. All that is required of users is to instantiate an instance of the F90Array class. All issues relating to memory management and the creation of the Fortran 90 array counterpart are handled in the constructor and destructor methods of the F90Array class (see the Appendix).

To make memory management more user-extensible, the F90Memory template class has been provided, which contains two static methods, malloc and free. In general, these methods just allocate/deallocate memory by using the C++ built-in memory management facilities (e.g. new/delete or malloc/free). For every type/rank combination for which an instance of F90Array exists, we produce a specialization of the F90Memory class, whose static functions call the appropriate F90MallocXY and F90FreeXY functions, and thereby allocate and free a Fortran array. This scheme makes use of external polymorphism and, thus, is very extensible [14].

A positive side-effect of using templates for array management is that the type and the rank of the array become visible to the C++ compiler. We can therefore provide indexing operators for the array class that return elements of the correct type and thereby help the C++ compiler in performing static type checking.

When developing algorithms, and when using the indexing operator, one must be aware that Fortran arrays are laid out in column-major order. Rather than managing the details of this directly, we simply provide an interface to the Blitz++ array library [15] which provides this functionality. Blitz++ is an expression template C++ library that provides very good support for high-performance array computations in C++. Blitz++ allows for a Fortran-like array syntax within C++, without introducing abstraction penalties during runtime.

4.1 Arrays Passed as Procedure Parameters

The policy for passing arrays as parameters to procedures is that the C++ stub interfaces generated by CHASM pass array proxies by reference and by value. Specifically, three cases exist: (1) an F90Array reference is returned as a function parameter when intent out is specified for the Fortran array; (2) an F90Array reference is passed as a function parameter when intent in is specified; (3) an F90Array is returned (by value) when a Fortran function returns an array.

An example of the first case is shown in the following code section (where the first line is a function prototype):

```
void returnAnArray(F90Array<int,2> &);

F90Array<int,2> A;
returnAnArray(A);
```

In this case, the default constructor creates an empty array, that is then filled by the returnAnArray() call. The second case shown below is similar, except that an array is first

constructed and memory allocated by calls to Fortran from C++:

```
void useAnArray(F90Array<int,2> &);

long lower[2] = {1, 1};
long upper[2] = {100, 100};

F90Array<int,2> B(lower, upper);
useAnArray(B);
```

The final code segment returns an array proxy by value:

```
F90Array<int,2> returnAnArray();

F90Array<int,2> C = returnAnArray();
```

5.0 Summary

CHASM is a set of software tools that help to bridge the divide between C++ and Fortran 90. These tools first parse Fortran source code to discover a list of procedures. A pair of C++ stub and Fortran skeleton functions is then generated for each Fortran procedure found. The C++ stub functions are used primarily to hide the symbol-naming convention of the specific Fortran compiler, and to allow the passing of C++ arrays by reference to a Fortran skeleton procedure expecting an integer handle representing the array. The Fortran skeleton functions are used to transform the integer array handle into a fully typed Fortran 90 array, and to make the call to the original Fortran procedure.

The CHASM tools must also generate additional functions and ancillary data structures to handle the sharing of arrays between C++ and Fortran 90. For each combination of array element type and array rank, a set of memory allocation and deallocation routines must be generated, in addition to a set of globally accessible storage arrays that associate array pointers with integer handles.

In addition, a C++ array template class is used to provide a consistent and easy to use interface for the sharing of arrays between Fortran and C++. Users need not be concerned about the underlying details, but simply use normal C++ constructors and indexing mechanisms with shared arrays. Meta information about the shared array, including the array rank and size, can also be accessed from C++.

The primary focus of this paper is on the sharing of arrays between Fortran 90 and C++ and on the calling of Fortran procedures from C++. Future work must address the calling asymmetry and consider the calling of C++ methods from Fortran. In addition, the automatic generation of proxy classes in both languages (to mimic the concrete classes or user defined types in the other language) should also be considered.

6.0 Appendix

The ArrayDesc class is used to access Fortran array descriptor information from C++. Public interfaces for this class are given below:

```
class ArrayDesc {
 public:

  enum DataType {
    Int1 = 1, Int2, Int4, Int8, Logical1, Logical2, Logical4,
    Logical8, Real4, Real8, Real16, Complex4, Complex8, Character
  };

  //
  // Get accessor functions
  //

  int rank() const;
  DataType dataType() const;
  void* address() const;
  void getDimensionInfo(int r, long& lower, long& upper) const;

  //
  // Set accessor functions
  //

  void rank(int r);
  void dataType(DataType dt);
  void address(void* addr);
  void setDimensionInfo(int r, long lower, long upper);
};
```

The F90Array class is used to provide access to Fortran 90 arrays from C++. Public interfaces for this class are given below:

```
template <typename ELEMENT_TYPE, int RANK>
class F90Array {
 public:

  //
  // Constructor and destructor methods.  The default constructor
  // creates an empty array to be filled in later by a call to a
  // Fortran function returning an array.  The lower and upper
  // parameters to the constructor are arrays containing the lower
  // and upper index for each dimension.
  //

  F90Array();
  F90Array(long* lower, long* upper);
  ~F90Array();

  //
  // Array indexing operators
  //
```

```
  ELEMENT_TYPE & operator[](const int& x);
  const ELEMENT_TYPE & operator[](const int&) const;
  operator ELEMENT_TYPE*();
  operator const ELEMENT_TYPE*() const;

  //
  // Accessor methods returning meta information about the array
  //

  int rank() const;
  ArrayDesc::DataType dataType() const;
  ELEMENT_TYPE* address() const;
  void getDimensionInfo(int r, long& lower, long& upper) const;
};
```

7.0 References

[1] A. Korosmezey, C.E. Rasmussen, T.I. Gombosi, and B. van Leer. Transport of Gyration Dominated Space Plasmas of Thermal Origin II: Numerical Solution. J. Computational Phys., 109, 16-29, 1993.

[2] K.G. Powell, P.L. Roe, T.J. Linde, T.I. Gombosi, and D. L. DeZeeuw. A Solution-Adaptive Upwind Scheme for Ideal Magnetohydrodynamics. J. Computational Phys., 154, 284-309, 1999.

[3] J.C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith, and J.L. Wagener. Fortran 90 Handbook, Complete ANSI/ISO Reference. McGraw-Hill, 1992.

[4] P.S. Pacheco. Parallel Programming with MPI. Morgan Kaufmann, 1997.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, version 1.1, June 12, 1995. Available by anonymous ftp from ftp.mcs.anl.gov.

[6] K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. Proceedings of SC2000: High Performance Networking and Computing Conference, November 2000.

[7] S. Shende, A.D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel, Scientific Applications Using C++. Proceedings of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134-145, August 1998.

[8] R.D. Rivenburgh, C.E. Rasmussen, K.A. Lindlan, B. Mohr, and P.H. Beckman. Automatic Generation of Perl Extensions to C++ and Fortran 90 Class Libraries. O'Reilly Open Source Software Convention, July 2000.

[9] Mutek. Fortran 90 Front End Documentation. http://www.mutek.com/, 1999-2001.

[10] Edison Design Group. Compiler Front Ends for the OEM Market. http://www.edg.com/, 1998-2001.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[12] Object Management Group. The Common Object Request Broker: Architecture and Specification. John Wiley & Sons, 1992.

[13] T.J. Bowbray and T. Brando. Interoperability and CORBA-based Open Systems. Object Magazine, September 1994.

[14] C. Cleeland. External Polymorphism: An Object Structural Pattern for Transparently Extending C++ Concrete Data Types. http://www.cs.wustl.edu/~cleeland/papers/External-Polymorphism/, 1996.

[15] T. Veldhuizen. Arrays in Blitz++. Proceedings of the 2nd International Symposium on Computing in Object-oriented Parallel Environments, 1998.