ISSN 1500-6050

Oslo Scientific Computing Archive

Report 1998-3

Writing C++ Interfaces to FORTRAN Packages

D. Calhoun H. P. Langtangen

June 20, 1998

Aims and scope: Traditionally, scientific documentation of many of the activities in modern scientific computing, like e.g. code design and development, software guides and results of extensive computer experiments, have received minor attention, at least in journals, books and preprint series, although the the results of such activites are of fundamental importance for further progress in the field. The Oslo Scientific Computing Archive is a forum for documenting advances in scientific computing, with a particular emphasis on topics that are not yet covered in the established literature. These topics include design of computer codes, utilization of modern programming techniques, like object-oriented and object-based programming, user's guide to software packages, verification and reliability of computer codes, visualization techniques and examples, concurrent computing, technical discussions of computational efficiency, problem solving environments, description of mathematical or numerical methods along with a guide to software implementing the methods, results of extensive computer experiments, and review, comparison and/or evaluation of software tools for scientific computing. The archive may also contain the software along with its documentation. More traditional development and analysis of mathematical models and numerical methods are welcome, and the archive may then act as a preprint series. There is no copyright, and the authors are always free to publish the material elsewhere. All contributions are subject to a quality control.

Oslo Scientific Computing Archive 1998-3 Revised June 20, 1998

Title

Writing C++ Interfaces to FORTRAN Packages

Contributed by

D. Calhoun
H. P. Langtangen

Communicated by

Are Magnus Bruaset

Contents

This report should be referenced as shown in the following BibTeX entry:

```
@techreport{OSCA1998-3,
  author = "D. Calhoun and H. P. Langtangen",
  title = "Writing C++ Interfaces to FORTRAN Packages",
  type = "Oslo Scientific Computing Archive",
  note = "URL: http://www.math.uio.no/OSCA; ISSN 1500-6050",
  number = "\#{}1998-3",
  year = "June 20, 1998",
}
```

Writing C++ Interfaces to FORTRAN Packages

D. Calhoun[1] **H. P. Langtangen**[2]

Abstract

The report starts with a summary of the mechanics of calling FORTRAN from C++. Thereafter, a convention for writing C++ interfaces to FORTRAN packages is described. In particular, we demonstrate how to program user defined functions, required by the FORTRAN package, as virtual functions in C++.

1 Introduction

Scientists and engineers have already realized the benefits of object-oriented programming in their code development. Due to computational efficiency reasons, object-oriented constructs are usually restricted to higher-level administration code, while the most CPU-time intensive computations take place in low-level code involving standard loops and simple array data structures, which are easily reconginzed for optimization by the compiler. Such program design opens up for the possibility of using existing high-quality FORTRAN packages as the low-level code. C++ has emerged as the standard language for object-oriented programming in numerical applications. Although the C++ syntax is in many respects quite different from FORTRAN, the two languages apply practically the same basic data types, memory addressing and calling conventions. For this reason, it is easy to make use of existing FORTRAN code when developing modern object-oriented numerical applications in C++.

This short report details briefly the mechanics of how to call FORTRAN from C++, and then goes in to detail on how to create a C++ "wrapper" for existing FORTRAN code. One of the key issues in the latter is how to

[1]Department of Applied Mathematics, University of Washington, USA. Email: calhoun@amath.washington.edu.

[2]Department of Mathematics, University of Oslo, Norway. Email: hpl@math.uio.no.

pass virtual functions defined in a C++ class to FORTRAN subroutines. Readers with experience in calling FOTRAN from C or C++ can skip the introductory sections and go directly to section 4 and read about the main idea of the present report.

It is assumed that the reader is familiar with both FORTRAN and C++, and has an understanding of the basic concepts in object-oriented programming. This report will contain references to classes in the Diffpack library [2]. While the reader may not be familiar with the specifics of the Diffpack classes, the context in which they will be referred should make their functionality apparant. Appendices will provide any essential details necessary for understanding the code presented in this report.

2 The mechanics of calling FORTRAN from C++

Calling FORTRAN functions from C or C++ and vice-versa is straightforward under the UNIX operating system and the technicalities are usually well-documented (if not always easy to find) in language reference manuals and textbooks [1]. Presented here are the essential features that users should be aware of when calling FORTRAN from C or C++.

2.1 Linkage

In order to accomodate the essential object-oriented feature of polymorphism, C++ allows the user to create several functions with the same name [1]. These functions differ because they either 1) are defined in distinct classes (usually in classes which are derived from a common base class) or 2) have argument lists, or "signatures", that differ in the type of data that is passed to the function and returned from the function. Ultimately, however, the C++ compiler must be able to distinguish these identically named functions, and does so by constructing internal names for these functions. While the details are not important, the names are formed by encoding the information about the class in which the function is defined and the argument list for the function. This defining of internal names for C++ functions is often referred to as "name mangling".

When calling FORTRAN from C++, it is essential that names for the FORTRAN subroutines, which will be prototyped in C++ header files, *not* be mangled. To prevent the usual name mangling carried out by the C++ compiler, declarations of FORTRAN subroutines in C++ header files must be identified as requiring either FORTRAN or C linkage. This is done by using the extern "FORTRAN" or extern "C" wrapper:

```
// Use FORTRAN linkage.  Not supported by all compilers
extern "FORTRAN" {
// Declare prototype for FORTRAN subroutine here
}
```

```
// Use C linkage.  Most commonly used.
extern "C" {
// Declare prototype for FORTRAN subroutine here
}
```

The reader should note that not all compilers support the FORTRAN style linkage and so C style linkage is most commonly used.

2.2 Declaring FORTRAN subroutines in C++ program files

A discussion of how the FORTRAN subroutines should be declared ultimately involves describing how to call the FORTRAN subroutine from C++, but for now, the two issues will be discussed separately. To prototype a FORTRAN subroutine in C++ header files, one must first know that many (but not all!) compilers require the use of an underscore suffix on FORTRAN subroutine names that are declared in C++. So, for example, if the user wishes to declare the use of the Lapack FORTRAN subroutine SGTSV for solving a tridiagonal system, the subroutine should actually be declared as `sgtsv_`. The use of lower-case in the name is important. Declaring the FORTRAN subroutine using the name `SGTSV_` (or `SGTSV`) will cause the linker to report that the subroutine `SGTSV_` (or `SGTSV`) is unresolved.

The argument list: Passing references to data. The key difference between FORTRAN and C/C++ is that all arguments to subroutines in FORTRAN are passed by reference, whereas in C/C++ data can be passed to functions by value or by reference. In fact, since FORTRAN subroutines have no formal output arguments, any results produced by the FORTRAN subroutine are recorded on variables passed into the subroutine as input. The C/C++ function on the other hand, will, without use of special syntacial structures such as pointers, only expect to operate on a local copy of the data it requires. Any changes made to this data is lost once the function goes out of scope. To accomodate this difference in the way in which function arguments are handled by the two different languages, the user must pass address locations of data to the FORTRAN subroutine, rather than the data itself. This is accomplished by the use of either pointer variables or references. For example, a FORTRAN subroutine ADD which adds two integers and stores the result in a third integer variable could have a prototype which looks like

```
extern "C"
{
  void add1_(int* a, int* b, int* a_plus_b);
}
```

This is the standard way of prototyping a FORTRAN function in C. A more sophisticated approach to prototyping implicitly documents whether

parameters are input or output data and makes use of references instead of pointers. By using the `const X&` for input data that is not to be changed by the subroutine, and `X&` for variables on which output results are to recorded, one makes the function prototype more informative. With this in mind, the above prototype could be written like this in C++:

```
extern "C"
{
  void add2_(const int& a, const int& b, int& a_plus_b);
}
```

Here, variables `a` and `b` are input data and are not to be changed, whereas output variable `a_plus_b` should be changed by the subroutine.

It should be noted that by simply declaring parameters as `const` in the C++ header files, the user is not protected from changes that the FORTRAN subroutine may make to these variables. The FORTRAN compiler has no way of knowing that variables passed to it have been declared as `const` and will allow redefinition of such variables. Moreover, any such redefinition will alter the original data, possibly destroying the integrity of the that data. The prototyping done using the `const` declarator should be viewed as documentation only.[3]

2.3 Calling the FORTRAN subroutine

Calling the FORTRAN subroutine from within a C++ program is straightforward, although it does depend on how the function was prototyped. For example, the FORTRAN function `add1` above can be called using the following short program:

```
void main()
{
  int a = 1, b = 5, a_plus_b;
  add1_(&a, &b, &a_plus_b);
  cout << "Result is : " << a_plus_b << endl;
}
```

Our `add2` function allows a nicer syntax:

```
void main()
{
  int a = 1, b = 5, a_plus_b;
  add2_(a, b, a_plus_b);
  cout << "Result is : " << a_plus_b << endl;
}
```

[3]Recall that in C++, the compiler issues an error if a `const` variable is changed inside a function.

2.4 Passing constants

Constants may be passed to FORTRAN subroutines in the usual way. For example, in this call to `add2`, constants are passed instead of parameters:

```
void main()
{
   int a_plus_b;
   add2_(2, 3, a_plus_b);
   cout << "Result is : " << a_plus_b << endl;
}
```

2.5 Passing data arrays

By using the `new` command, users can create dynamic arrays of data easily in C++ (at run-time), and then pass this data to FORTRAN to be used as input data or output data. For example, the following program can be used to add to vectors together:

```
#include <iostream.h>

extern "C"
{ void addVec_(const float v1[], const float v2[], const int& n, float v3[]); }
// alternative:
// void addVec_(const float* v1, const float* v2, const int& n, float* v3); }

void main()
{
  int n = 10;
  float* v1 = new float[n];
  float* v2 = new float[n];
  float* v3 = new float[n];
  for(int i = 0; i < n; i++)   {
     v1[i] =  i;  v2[i] = -i;  }
  addVec_(v1, v2, n, v3);
  for(i = 0; i < n; i++)
     cout << v3[i] << " ";
  cout << endl;
}
```

When passing arrays of more than one dimension to FORTRAN, the user should be aware of the data storage differences between FORTRAN and C++. In FORTRAN two dimsional arrays, for example, are stored in "column-major" format, whereas in C++ data is stored in row major format. As a consequence, the user must pass transposed data arrays to the FORTRAN compiler. Since it is assumed that the user is familar with, or has access to, C++ matrix libraries which can perform the transposition, the details are not given here.

2.6 Data type compatibility

In the above discussion of calling FORTRAN subroutines, it is assumed that the argument list for the FORTRAN subroutine is compatible with that in the C++ header file. The C++ compiler will be able to catch inconsistencies between how a function is prototyped and how it is called, but cannot catch any data inconsistencies between the C++ prototype and the actual definition of the FORTRAN subroutine. For this reason, the user must be especially careful in making sure that, for example, doubles in C++ are passed to `REAL*8`'s in FORTRAN, that floats in C++ are passed to `REAL*4`'s in FORTRAN, and that memory for any array is passed correctly to the FORTRAN routine. The user should consult language references for the exact memory requirements of primitive types in C++ and FORTRAN. In particular, one should be especially careful with the primitive C/C++ type `int`, as it may occupy 4 or 8 bytes, depending on the compiler used.

2.7 An Example: Solving a tridiagonal system

One obvious reason to make use of existing FORTRAN code in scientific applications written in C++ is that most public domain scientific software is written in FORTRAN. In particular, Lapack (an extensive library for numerical linear algebra) is considered the industry standard for such libraries. This simple example illustrates how one can create a class that makes use of Lapack to solve tridiagonal linear systems. The name of the appropriate Lapack routine to use is `DGTSV`. Instead of using primitive C arrays to hold the matrix data, we apply (as usual in C++ code) more user friendly class abstractions for arrays. Here, we assume that class `Vec` is an encapsulation of a standard one-dimensional C array of doubles. The function

```
double* Vec::getPtr0()
```

returns the pointer to the first element of the underlying C array. This pointer is essential for communicating with FORTRAN. Moreover, the number of entries in the C array is given from

```
int Vec::size()
```

Here is an extraction of the code:

```
// Declararation of FORTRAN subroutine
extern "C" {
    void dgtsv_(const int& N, const int& NRHS,
                const double DL[], const double D[],
                const double DU[], const double B[],
                const int& LDB, int& INFO);
}

// Class for storing a tridiagonal matrix
```

```
class TriDiagMatrix
{
private:
    Vec main_diag, upper_diag, lower_diag;
public:
    void solve (Vec& b)
    {
        // Solve a tridiagonal system Tx=b, in double precision.
        // From Lapack User's Guide (page 153)
        // SUBROUTINE DGTSV(N,NRHS,DL,D,DU,B,LDB,INFO)
        // N         : Size of the matrix
        // NRHS      : Number of right hand sides
        // DL, D, DU : 3 diagonals
        // B         : right hand side vector. WILL BE overwritten!
        // LDB       : leading dimension of the array B.
        // INFO      : Output information

        // Call Lapack routine
        int info;
        dgtsv_(main_diag.size(),1,lower_diag.getPtr0(),main_diag.getPtr0(),
               upper_diag.getPtr0(),b.getPtr0(),b.size(),info);
        // process info if desired...
    }
   // additional utility functions...
};
```

3 Passing functions as arguments to FORTRAN routines

Passing global functions defined in C to a FORTRAN subroutine is straightforward and is illustrated in the following example. Here, a FORTRAN subroutine called `INTEGRATE` requires a function, in addition to endpoints and output data, as arguments. The subroutine `INTEGRATE` calls the input function and evaluates the integral. Note that the actual name of `INTEGRAL`, as seen from C, C++ and the operating system, is `integrate_`

```
// ------------- C++ file --------------------------------
#include <iostream.h>
#include <math.h>

typedef void (*CFunc_Ptr)(const double& x, double& value);

extern "C"
{
  integrate_((CFunc_Ptr) f, const double& a, const double& b, double& result);
}

void userFunction (const  double& x, double& value)
{ value = exp(x) + 1; }

void main()
```

```
{
  double a = 0, b = 1, result;
  integrate_((CFunc_Ptr) userFunction, a, b, result);
  cout << "Integral is : " << result  << endl;
}

C ----------------- FORTRAN SUBROUTINE -------------------------
C Use trapazoidal approximation to integral
C
      subroutine INTEGRATE(f,A,B,Y)
      REAL*8 Y, A,B,END1,END2
      EXTERNAL f

      CALL f(A,END1)
      CALL f(B,END2)
      y = 0.5*(END1 + END2)*(B - A)

      RETURN
      END
```

The user should note the use of the `EXTERNAL` declaration in the FORTRAN subroutine.

3.1 Passing member functions to a FORTRAN subroutine

To take full advantage of the benefits of object oriented programming, the user will no doubt want to create classes in which user specified functions are defined as member functions in user-defined classes. In many instances, the user may then wish to pass these functions to a FORTRAN subroutine. However, these functions cannot be directly passed to a FORTRAN subroutine, because they can only be called by an instance of the class to which they belong. Since FORTRAN knows nothing about instances of C++ objects, FORTRAN can only call C++ functions which are defined as global. To work around this apparant difficulty, a global function which can be passed to the FORTRAN routine is defined to call, by means of a global pointer to an instance of the class whose member function is desired, the member function of interest. This is illustrated most succinctly in the following code. Here, the global function calls, by means of the global pointer to an instance of class `A`, a member function of the class `A`:

```
// C++ program file
#include <iostream.h>

class A;
A* A_ptr;  // global pointer

typedef void (*CFunc_Ptr)(const double& x, double& value);

// FORTRAN subroutine
extern "C"
```

```
{
  integrate_((CFunc_Ptr) f, const double& a, const double& b, double& result);
}

// C-style global function passed to FORTRAN
void CFunction_square(const double& x, double& value)
{ value = A_ptr->square(x); }

class A
{
  // some data - if necessary...
public:
  double square (double x) { return x*x; }
}

void main()
{
    A* anA = new A;
    A_ptr = anA;
    double a = 0, b = 1, result;
    integrate_((CFunc_Ptr) CFunction_square, a, b, result);
    cout << "Result is " << result << endl;
}
```

The above will work in many cases where the user does not have a complicated inheritance tree to work with, and therefore is not working with virtual functions. The function `square` may of course make use of any instance variables or member functions in the class `A`.

3.2 Passing virtual functions to FORTRAN

To take advantage of polymorphism, the user will no doubt want to create base classes with virtual functions which are later defined in derived classes. The functions in the derived classes are then passed to the FORTRAN subroutine in the manner described above. This too is quite straightforward, since the interface to the class is defined in a base class, and so a global pointer to the base class can be used in the global C-style function that must be passed to FORTRAN.

Suppose that the user wishes to create a class `Integral`, which has as instance variables an interval, described by the endpoints (a, b). By means of the overloaded parenthesis operator, instances of `Integral` can behave as a functional which takes as an argument a functor (an object which behaves as a function) and returns the evaluation of the definite integral of the function over the interval (a, b). The class might look something like this:

```
class Integral
{
private:
  double fA, fB; // Endpoints of the definite integral.
public:
```

```
  Integral(double a, double b) { fA = a; fB = b;}
  // This function makes a call to the FORTRAN subroutine INTEGRATE
  double operator() (MathFunctions* f)
};
```

Here, the class `MathFunctions` is an abstract base class with a pure virtual member function `func`. This function is defined in subclasses that are derived from this class. The user creates instances of `MathFunctions` and passes them to an instance of the class `Integral`, as the following code fragment illustrates:

```
// C++ code fragment
void main()
{
    // Integrate over the inteval (0,1).
    Integral I(0,1);

    // Create instance of subclass of MathFunctions, math_pow
    double p = 2.0;
    math_pow f1(p);   // f1(x) = x^p
    cout << " I(f1) = " << I(&f1) << endl;

    // An instance of another subclass of MathFunctions, math_linear
    double a = 1.0, b = 1.0;
    math_linear f2(a,b);  // f2(x) = a*x + b;
    cout << " I(f2) = " << I(&f2) << endl;
}
```

The class `MathFunctions` has two essential tasks. One, it defines a common interface which can be called by the global function which is eventually passed to the FORTRAN subroutine. Second, it must assign a value to a global pointer to an instance of the class `MathFunctions`. With this in mind, the definition of `MathFunctions` is quite simple. This time, we will avoid the use of a global function pointer by using a static variable declared in the class `MathFunctions`:

```
// More C++ code fragments
class MathFunctions
{
public:
  friend class Integral;
  virtual double func(double x) = 0;
  static MathFunctions* MathFunctions_class_ptr;
protected:
  virtual void init() { MathFunctions_class_ptr = this; }
};
```

The use of the static variable here only declares it as a member of the class `MathFunctions`; it does *not* allocate any storage for the pointer. This must be done in the following global statement:

```
// Allocate storage for the static member of MathFunctions:
MathFunctions* MathFunctions::MathFunctions_class_ptr = NULL;
```

Derived classes then take the following form:

```
// User defined class
class math_pow : public MathFunctions
{
  double p;
public:
  math_pow (int p_) { p = p_; }
  // warning from some C++ compilers: math_pow (int p) { this->p = p; }
  virtual double func (double x) { return pow(x,p); }
};

// User defined class
class math_linear : public MathFunctions
{
  double a,b;
public:
  math_linear (double a_, double b_) { a = a_; b = b_; }
  virtual double func (double x)   { return fSlope*x + fConstant; }
};
```

The global function passed to the FORTRAN subroutine `INTEGRATE` is defined as:

```
// Type definition for function that is passed to FORTRAN
typedef void (*CFunc_Ptr)(const double& x, double& value);

// Math function of type CFunc_Ptr.
void global_mathfunc (const double& x, double& value)
{
  value = MathFunctions::MathFunctions_class_ptr->func(x);
}
```

Here, there is not problem in calling `func` using the global pointer

`MathFunctions_class_ptr`

since `func` is defined in the interface for `MathFunctions`.

Finally, the definition of

`Integral::operator()(MathFunctions* f)`

should now be apparant:

```
double Integral:: operator() (MathFunctions* f)
{
  double result;
  f->_initializePtrs();
  integrate_((CFunc_Ptr) global_mathfunc, fA, fB, result);
  return result;
}
```

Note that there is nothing here that depends on knowledge of subclasses of `MathFunctions`.

4 A complete example

The above examples illustrate the main principles one needs to adhere to when calling a FORTRAN subroutine from C++. In particular, it was noted that virtual fuctions may be passed quite easily, with the help of global "helper" functions, to a FORTRAN subroutine.

In some instances, however, it may not be desirable to seperate the class which calls the FORTRAN subroutine, from the definition of the virtual functions. The virtual functions may require data only available in the functions which call the FORTRAN subroutines. The underlying principles described in this section are essentially the same as that described in the above approach, but the philosophy is slightly different.

To be more specific and illustrate the ideas further, we will now present a toy package in FORTRAN, called `SCL`, for solving the simple problem

$$u_t + f(u)_x = 0, \quad x \in (0, L)$$

by the upwind finite difference scheme for $0 < t \leq T$. The initial condition reads $u(x, 0) = g(x)$. Let u_i^k be the numerical approximation to $u((i-1)\Delta x, k\Delta t)$, where Δx and Δt are the space and time step, respectively, and $i = 1, \ldots, n$ and $k = 1, \ldots, T/\Delta t$. The numerical algorithm then takes the following form.

- Set initial condition for u_i^0.
- Compute u_i^k from the explicit scheme

$$u_i^k = u_i^{k-1} - \frac{\Delta t}{\Delta x}\left(f(u_i^{k-1}) - f(u_{i-1}^{k-1})\right)$$

  for $i = 2, \ldots, n$ and $k = 1, \ldots$.

4.1 The FORTRAN package interface

We assume that the user must supply a function `user_u0` for setting the initial conditions. Moreover, the specific form of the flux function $f(u)$ is provided in another user defined function `user_flux`. Calling the function `scheme` results in computing $u(x, T)$.

```
subroutine u0 (u_prev, n)
integer n
double precision u_prev(n)

double precision function flux (u_value)
double precision u_value

subroutine scheme (u, u_prev, n, dx, dt, T, user_u0, user_flux, dbg)
integer n, dbg, i
double precision u(n), u_prev(n), dx, dt, T, user_flux
external user_u0, user_flux
```

The complete listing of the FORTRAN functions appears in appendix A.

In C++ it would be natural to have a class where the input data, like `u`, `u_prev`, `dx`, `dt` etc., are class members and where the user defined functions are virtual functions. Moreover, we include a `scan` function for reading the input data and allocating dynamic arrays (`u` and `u_prev`). For example,

```
class SCL
{
protected:
  double* u; double* u_prev;
  double  dt, dx, L, T;
  int n;
public:
  virtual void u0 () = 0;
  virtual void flux (double& value) = 0;
  static SCL* f2cpp;     // global pointer for virtual function calls
  virtual void scan ();  // read dt, dx etc., allocate u and u_prev, set f2cpp
  void solveProblem ();  // call FORTRAN function scheme
};
```

A specific set of user functions can then be implemented in a subclass for a particular application:

```
class MyProblem : public SCL
{
public:
  virtual void u0 ();
  virtual void flux (double& value) { return value; } // linear flux
};

void MyProblem:: u0 ()
{
  u_prev[0] = 1.0;
  for (int i = 1; i < n; i++)
    u_prev[i] = 0.0;
}
```

4.2 An alternative C++ class

The C++ class `SCL` applied primitive C data structures. In a C++ program one will often use objects at higher abstraction levels. Let us now, for demonstration purposes, apply high-level C++ abstractions for the computational grid, as well as for the scalar field u at the current and previous time level. The abstractions are taken from Diffpack [2]. That is, $u(\cdot, t)$ is represented as a scalar, spatial, finite difference field of type `FieldLattice`. Roughly speaking, a `FieldLattice` object contains a specification of a grid and an array of the point values of the field.

```
class FieldLattice
```

```
{
  Handle(GridLattice)    mesh;
  Handle(ArrayGen(real)) vec;
public:
  ArrayGen(real)& values() { return *vec; }  // user's access to point values
  // indexing functions
  // interpolation functions etc.
};
```

The construction `Handle(X)` is just a smart pointer to an `X` object. The `GridLattice` class holds Δx, n and similar quantities for a general d dimensional uniform lattice. The `ArrayGen(real)` class represents a general array of `real` numbers (`real` is here identical to `double`), with single or multiple indices. The storage of `ArrayGen(real)` is compatible with multi-dimensional arrays in FORTRAN, that is, only a long vector is used, and multiple indices are transformed to single indices in overloaded `operator()` functions.

When communicating with FORTRAN, one can of course not use the `ArrayGen(real)` abstraction directly. The only array type that FORTRAN understands, is a basic C array transferred by a pointer to the first entry of the array. Class `ArrayGen(real)` applies a `real* A` pointer internally for storing the array entries. A function `getPtr0()` returns the address of the first entry and can be used for transferring an `ArrayGen(real)` array to FORTRAN code. Here we see how important it is that high-level C++ abstractions use basic C data structures to actually hold the data. Fancy private data structures, e.g. lists of memory blocks, in C++ classes makes efficient communication with FORTRAN modules difficult.

The definition of class `DpSCL` can look like this:

```
class DpSCL
{
protected:
  Handle(GridLattice)  grid;   // holds domain (0,L), n and grid spacing
  Handle(FieldLattice) u;      // u at the current time level
  Handle(FieldLattice) u_prev; // u at the previous time level
  real                 dt, T;  // problem parameters
public:
  static DpSCL*        f2cpp;  // global pointer for virtual function calls

  // user defined functions required by the FORTRAN package:
  virtual void u0 () = 0;
  virtual real flux (real u_value) = 0;

  virtual void scan ();  // read L, n etc and initialize data structures
  void solveProblem ();  // use FORTRAN package to compute u(x,T)
  void resultReport ();  // print final solution u(x,T)
};
```

We can understand why the `u0` function does not need any arguments. This is because the `DpSCL` class has all the data that are necessary in `u0`. The purpose of `u0` is to fill `u_prev` with proper data.

The `scan` function reads parameters, like n, L and Δt, and allocates the data structures `grid`, `u` and `u_prev`. The `solveProblem` function just calls the `scheme` function in the FORTRAN package.

In the `DpSCL` class we also include a static pointer `DpSCL* f2cpp` that can be used to invoke the user defined virtual functions anywhere in the code (`f2cpp` is hence similar to a standard global variable). The `f2cpp` pointer must be initialized in, e.g., the `DpSCL::scan` function (in that function it should be set to point to the `this` variable).

To solve a specific problem, derive a subclass where you implement the `u0` and `flux` functions, and additional data structures (and a redefined `scan` function to initialize them) if desired. An example is given in appendix C.

4.3 The FORTRAN interface as seen from C++

The only FORTRAN function that is called by the C++ code is `scheme`. This function must be prototyped in an `extern "C"` statement. As already mentioned, user defined virtual functions need to be called from "shell" functions with a plain C signature. Here we introduce two such shell functions; `u0_i` and `flux_i`. These functions must take the same parameters as expected in the FORTRAN routines `user_u0` and `user_flux` and call the user defined virtual functions in `DpSCL` by means of the `DpSCL::f2cpp` pointer. In the function `u0_i` one can make use of the fact that the transferred `u_prev` array is identical to the internal array in the `u_prev` field in the `DpSCL` class. That is why we do not need to transfer the array to `DpSCL::u0`.

The relevant `extern "C"` declarations, the definition of class `DpSCL` and the definition of the shell functions are given in detail in appendix C. Here we just illustrate the main ideas.

```
// make function pointers (needed as arguments in call to scheme):
typedef void (*u0_ptr) (double u_prev[], const int& n);  // for u0_i
typedef double (*flux_ptr) (const double& u_value);      // for flux_i

extern "C"
{
  // FORTRAN functions called from C++
  double scheme_ (double u[], double u_prev[], const int& n,
                  const double& dx, const double& dt, const double& T,
                  u0_ptr user_u0, flux_ptr user_flux, const int& dbg);

  // C interface for virtual C++ functions called from FORTRAN:
  void u0_i_ (double u_prev[], const int& n);
  double flux_i_ (const double& u_value);
}

DpSCL* DpSCL::f2cpp = NULL;  // to be initialized in DpSCL::scan

void u0_i_ (double u_prev[], const int& n) { DpSCL::f2cpp->u0(); }
double flux_i_ (const double& u_value) { return DpSCL::f2cpp->flux (u_value); }
```

Next, we list the `DpSCL::solveProblem` function to see how we can call FORTRAN using the high-level abstractions `FieldLattice` and `GridLattice`.

```
void DpSCL:: solveProblem ()
{
  scheme (u->values().getPtr0(),  // ptr to first C array entry
          u_prev->values().getPtr0(), u->values().size(),
          grid->Delta(1),         // dx
          dt, T, u0_i_, flux_i_, 1);
}
```

A particular set of initial conditions and flux function can be implemented in class `MyProblem`:

```
#include <DpSCL.h>

class MyProblem : public DpSCL
{
public:
  virtual void u0 ();
  virtual real flux (real u_value);
};

void MyProblem:: u0 ()
{
  u_prev->fill (0.0);
  u_prev->values()(1) = 1;   // u_1^0 = 1
}

real MyProblem:: flux (real u_value)
{
  return u_value;   // linear flux
}

int main (int nargs, const char** args)
{
  initDiffpack (nargs, args);   // standard initialization of Diffpack programs
  MyProblem problem;
  problem.scan();
  problem.solveProblem();
  problem.resultReport();
}
```

5 Conclusions

We have demonstrated that it is possible to write a *general* interface class to a FORTRAN package. User defined functions for a particular application can be implemented in a subclass of the interface class. Hence, different applications have short codes because they can reuse the interface and associated data structures that are required by the FORTRAN package. Moreover, our standard for writing applications makes it straighforward to integrate the solvers in other object-oriented frameworks for scientific programming.

There are numerous issues that have not been covered in this report. We briefly mention common blocks in FORTRAN, the new FORTRAN90 language which supports abstract data structures (but not virtual functions), and casting of pointers to member functions of subclasses to member functions of the base class.

A Complete listing of the FORTRAN package SCL

The numerical algorithm is implemented in a file `scheme.f`:

```
      subroutine scheme (u, u_prev, n, dx, dt, T,
     +                   user_u0, user_flux, dbg)
      integer n, dbg, i
      double precision u(n), u_prev(n), dx, dt, T, user_flux
      external user_u0, user_flux
      double precision time, mu

      call user_u0 (u_prev, n)
      u(1) = u_prev(1)
      mu = dt/dx
      do 20 time = dt, T, dt
         do 10 i = 2, n
            u(i) = u_prev(i) - mu*(user_flux(u_prev(i))
     +                           - user_flux(u_prev(i-1)))
 10      continue
C        debug output?
         if (dbg .ne. 0) then
            call udebug (u, n, time)
         endif
         do 20 i = 2, n
            u_prev(i) = u(i)
 20      continue
 30   continue
      T = time
      end

      subroutine udebug (u, n, t)
      integer n, i
      double precision u(n), t
      do 10 i = 1,n
         write(*,*) 't=',t,', u(',i,')=',u(i)
 10   continue
      end
```

For test purposes we will choose $f(u) = u$ and the initial condition $u_1^0 = 1$, $u_i^0 = 0$, $i = 2, \ldots, n$. If we let $\Delta t = \Delta x$, the pointwise numerical solution coincides with the analytical solution $u(x,t) = 1 - H(x-t)$, where $H(\xi)$ is the Heaviside function. These choices of f and $u(x,0)$ are implemented in the following user defined functions (file `fuser.f`):

```
      subroutine u0 (u_prev, n)
      integer n,i
      double precision u_prev(n)
      u_prev(1) = 1.0
      do 10 i = 2,n
         u_prev(i) = 0
```

```
 10   continue
      end

      double precision function flux (u_value)
      double precision u_value
      flux = u_value
      return
      end
```

A simple main program to test the software can look like this:

```
      program upwind
      integer n
      external u0, flux
      double precision L, dx, dt, u(1000), u_prev(1000), T
      write(*,*) 'Give L: '
      read(*,*) L
      write(*,*) 'Give number of intervals: '
      read(*,*) n
C     n is from now on used as the number of points:
      n = n + 1
      dx = L/float(n-1)
      write(*,*) 'Give time step: '
      read(*,*) dt
      write(*,*) 'Give final time level: '
      read(*,*) T

      call scheme (u, u_prev, n, dx, dt, T, u0, flux, 1)
      call dump (u, n, T)
      end
```

B Using macros for portability

On some computer systems an underscore is appended to all FORTRAN subroutine and function names. Hence, if you have a `subroutine scheme`, its real name may be `scheme_` and this name must be used when calling the function from C or C++. Portability can easily be achieved by defining a macro `FORTRANname(X)` that equals `X` or `X_` according to the naming conventions on the particular computer system in question. For example,

```
#if defined(__DECCXX)               /* Dec Alpha? */
 #define FORTRANname(X) name2(X,_)  /* add underscore */
#else
 #define FORTRANname(X) X           /* no underscore on other machines */
#endif
```

In practice, one needs to include tests on other machine types in the if-statement (in fact, most systems will require an underscore - see the Diffpack file `macros.h` for the real definition of `FORTRANname`). Instead of just writing `scheme` in the C or C++ code, we write `FORTRANname(scheme)` to enable portability. This convention is used in the Diffpack example code in appendix C.

C Complete Diffpack listing of class `DpSCL`

As a reference for Diffpack programmers, we provide a complete listing of the class `DpSCL` and its member functions.

The interface is as given previously in the report, but the `FORTRANname` macro is used for portability.

```
#ifndef DpSCL_h_IS_INCLUDED
#define DpSCL_h_IS_INCLUDED
#include <FieldLattice.h>
#include <errors.h>
#include <macros.h>   // defines the macro FORTRANname

// make function pointers (needed as arguments in call to scheme):
typedef void (*u0_ptr) (double u_prev[], const int& n);  // for u0_i
typedef double (*flux_ptr) (const double& u_value);      // for flux_i

extern "C"
{
  // Fortran functions called from C++
  double FORTRANname(scheme) (double u[], double u_prev[], const int& n,
                              const double& dx, const double& dt,
                              const double& T,
                              u0_ptr user_u0, flux_ptr user_flux,
                              const int& dbg);

  // C interface for virtual C++ functions called from Fortran:
  void FORTRANname(u0_i) (double u_prev[], const int& n);
  double FORTRANname(flux_i) (const double& u_value);
}

class DpSCL
{
protected:
  Handle(GridLattice)  grid;   // holds (0,L), n and grid spacing
  Handle(FieldLattice) u;      // u at the current time level
  Handle(FieldLattice) u_prev; // u at the previous time level
  real                 dt, T;  // problem parameters
public:
  static DpSCL*        f2cpp;  // global pointer for calling virtual functions

  // user defined functions required by the Fortran package:
  virtual void u0 () = 0;
  virtual real flux (real u_value) = 0;

  virtual void scan ();  // read L, n etc and initialize data structures
  void solveProblem ();  // use Fortran package to compute u(x,T)
  void resultReport ();  // print final solution u(x,T)
};
#endif
```

The body of the member functions of class `DpSCL` and the interface functions in C are listed below.

```
#include <DpSCL.h>

DpSCL* DpSCL::f2cpp = NULL;

void FORTRANname(u0_i) (double u_prev[], const int& n)
{
```

```
  if (DpSCL::f2cpp == NULL)
    errorFP("u0_i","the DpSCL::f2cpp is not initialized.\n"
            "Set f2cpp to point to your simulator class!");

  // assume that the C++ class has the u_prev array and its length
  DpSCL::f2cpp->u0();
}

double FORTRANname(flux_i) (const double& u_value)
{
  return DpSCL::f2cpp->flux (u_value);
}


void DpSCL::scan ()
{
  // could use the menu system, here we just use command line arguments
  real L;
  initFromCommandLineArg ("-L", // command line option
                          L,    // variable to be initialized
                          1.0); // default value
  int n; initFromCommandLineArg ("-intervals",n,10); n++;  // n = no of points
  initFromCommandLineArg ("-dt",dt,0.1);
  initFromCommandLineArg ("-T",T,0.5);
  // example: app -L 2 -T 2 -dt 0.2

  DpSCL::f2cpp = this;  // important!

  grid.rebind (new GridLattice(1));
  // grid can be initialized with a string like "d=1 domain=[0,1] div=[1:20]"
  grid->scan (aform("d=1  domain=[0,%g]  div=[1:%d]",L,n));
  u.rebind (new FieldLattice(grid(), "u"));  // "u" is a fieldname
  u_prev.rebind (new FieldLattice(grid(), "u_prev"));
}

void DpSCL:: solveProblem ()
{
  FORTRANname(scheme) (u->values().getPtr0(),  // ptr to first C array entry
                       u_prev->values().getPtr0(),
                       u->values().size(),
                       grid->Delta(1),         // dx
                       dt, T,
                       FORTRANname(u0_i), FORTRANname(flux_i), 1);
}

void DpSCL:: resultReport ()
{
  u->values().print(s_o, "Final grid point values");
}
```

As a test example, one can compile the files `scheme.f`,[4] `DpSCL.h`, `DpSCL.cpp` and `MyProblem.cpp`, in the Diffpack subdirectory with name `C`, using the Diffpack `Make` script. Just typing `app` should produce a grid with 10 intervals and exact solution at the grid points.

[4] On some platforms, especially PC's running `linux`, it is convenient to transfer the FORTRAN file to C by running `f2c -A`.

References

[1] J. J. Barton and L. R. Nackman: *Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples.* Addison-Wesley, 1994.

[2] Diffpack World Wide Web home page: *http://www.nobjects.com/Diffpack*, 1998