Interfacing Fortran with Object-Oriented languages : some practical exemples

Éric Aubourg
CEA/DAPNIA/SPP
20/05/2000

Introduction

Two documents available in Planck DMS describe different ways to interface Fortran code with classes written in an object-oriented language like C++ or Java: *Planck HFI L2 Software Development Guidelines*, and *Going native seamlessly - calling C from Java*.
If one wants to allow Fortran developpers to interact with a DPC framework, or with the DMCI, an architectural choice must be made : either an object-oriented wrapper code handles the interaction with DPC/DMCI, and calls Fortran code, passing it relevant data, as the two documents mentioned above assume, or a main Fortran program calls the DMCI and the DPC using special Fortran interfaces.
The second solution might look easier to use for Fortran-only programmers, but might add extra burden to DPC/DMCI developers.
We will investigate those two solutions by solving a small problem using all the possible architectures, both with C++ and Java as the object-oriented language.

The problem

The DPC provides a very smart matrix class (`Mtx`), with a parallel, very efficient invesion algorithm. The DMCI provides a class to store and fetch this Matrix class from persistent storage (`MtxStore`).
A Fortran developper wants to implement, in Fortran, an algorithm that multiplies the values of the matrix by their colum index (starting from 1).
He wants to read a matrix from storage, apply his algorithm, invert the result, and store the final matrix back to storage.
The demonstration-only `Mtx` and `MtxStore` classes are given in annex.

Pure C++ solution

This solution is quite simple : C++ calls to DMCI read and store the matrix, and calls to DPC invert it. The algorithm is coded as a pure function (C-like).

```
/* Case 0 : the programmer learns C++ and codes its       */
/*          algorithm in C++ (myCalc)                     */
/*          he writes a C++ main program that handles     */
/*          standard Planck objects and call its code     */

/* Eric Aubourg, CEA/DAPNIA/SPP May 2000                  */
```

```
#include "mtx.h"
#include "mtxstore.h"

void myCalc(Mtx& a);

int main(int, char**) {
  // Instantiate persistence manager
  MtxStore* store = new MtxStore();
  // Read matrix
  Mtx A = store->fetch("MyMatrix");
  // Call my code to multiply matrix
  myCalc(A);
  // Inverse the matrix
  A.inverse();
  // Store the result
  store->store(A, "Result0");
}

void myCalc(Mtx& a) {
  for (int i=0; i<a.getNRows(); i++)
    for (int j=0; j<a.getNCols(); j++)
      a(i,j) *= (j+1);
}
```

Note that we will not handle the case of C as the non-object-oriented language. Is syntax is so close to C++ that a programmer who knows C can code the previous program without knowing the details of object-oriented programming. A small tutorial to DMCI and DPC would be enough for that.

Calling Fortran from C++

This process was described in the document *Planck HFI L2 Software Development Guidelines*. The Fortran routines are declared like C routines, but with an underscore at the end of the name —this behaviour is shared by all Unix Fortran compilers. All arguments are passed by reference (C pointers). Special care must be taken for strings: an extra integer parameter (passed by value) giving the length of the string must be added at the end of the parameter list for every string parameter.
The linking process is no big deal on most platforms. The easiest way is to link using the C++ compiler, adding fortran libraries. This way, a standard C++ main program will be bootstraped on program load, and the C++ link step can instanciate templates as needed.

File `case1.cc` : a C++ main program calling Fortran

```
/* Case 1 : the programmer codes its algorithm in FORTRAN */
/*          subroutine mycalc in for1.f                   */
/*          he writes a C++ main program that handles     */
/*          standard Planck objects and call its code     */

/* Eric Aubourg, CEA/DAPNIA/SPP May 2000                  */
```

```
#include "mtx.h"
#include "mtxstore.h"

extern "C" void mycalc_(float* data, int* nr, int* nc);

int main(int, char**) {
  // Instantiate persistence manager
  MtxStore* store = new MtxStore();
  // Read matrix
  Mtx A = store->fetch("MyMatrix");
  // Call my code to multiply matrix
  // Need to copy array because fortran arrays are
  // column-major - unless SOPHYA arrays are used
  float* fordat = new float[A.getNRows()*A.getNCols()];
  A.copyToForArray(fordat);
  int nr = A.getNRows(); // need copy to pass by reference
  int nc = A.getNCols();
  mycalc_(fordat, &nr, &nc);
  A.copyFromForArray(fordat);
  delete[] fordat;
  // Inverse the matrix
  A.inverse();
  // Store the result
  store->store(A, "Result1");
}
```

File `for1.f` : Fortran code called from C++

```
      subroutine mycalc(tab, nr, nc)
      implicit none
      integer nr,nc
      real*4 tab(nr,nc)
      integer i,j

      do i=1,nr
         do j=1,nr
            tab(i,j) = tab(i,j) * j
         enddo
      enddo

      return
      end
```

Calling C++ from Fortran

We need do define an interface to DPC and DMCI functions that can be called from Fortran. This implies that we must be able to manipulate C++ objects from Fortran (create them, read them from persistent storage, call DPC methods, store them, and destroy them).
No pointer system exists in Fortran, and even with Fortran 90 pointers would not really solve

our problem because we would not be able to do any kind of type-checking.
A solution can be found in the concept of “logical units” of Fortran: the C++ interface will generate a integer reference number for all objects manipulated from Fortran. The C++ interface will thus have to keep a map associating Fortran reference numbers to real objects.
In our example, we will write a C++ class providing those functionalities for the `Mtx` class. In the real world, one would either have to implement such a class for all classes that will be manipulated for Fortran, or try to build a generic mechanism based on the same principle, which is in theory doable but quite complex.
The linking process is much more complex than in the previous case. We would like to use the C++ linker to have templates instanciated when needed, and C++ special mechanisms (static object instanciation) activated, but our main program is in Fortran, and symmetry from the previous case would imply we should use the fortran compiler as a linking command.
The method that works usually best is to link with the C++ compiler, add fortran libraries, and add the Fortran main function. For instance, on Digital Unix using Digital compilers (cxx and f77) the following lines can be added to the makefile:

```
case2 : case2.o mtxintf.o mtx.o mtxstore.o
$(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@ -lUfor -lfor -lFutil \
          /usr/lib/cmplrs/fort/for_main.o
```

This is very system-dependant. No solution was found that allows using g++ as the C++ compiler on our Digital Unix system, either with f77 or g77, that did not require rebuilding a special Fortran-C++ runtime library.

File `mtxintf.h` : Fortran interface to DPC and DMCI

```
#ifndef MTXINTF_H
#define MTXINTF_H

#include "mtx.h"

/* Fortran interface to planck matrix objects             */
/* For demonstration only                                 */
/* Eric Aubourg, CEA/DAPNIA/SPP May 2000                  */

/* This class has to be written by Planck DPC developpers */
/* to support the model of fortran-only programs using    */
/* standards DPC objects                                  */

/* Requirements: we want to instanciate objects           */
/* from a main fortran code, call standard planck         */
/* methods on those objects, and access data from         */
/* fortran                                                */
/* We will access objects from Fortran using              */
/* integer values, in a way similar to logical            */
/* units, since there is no pointer in Fortran            */
/* The C++ code needs to keep a map of int -> object      */

/* A similar interface must be written for every          */
/* object, unless we attempt genericity                   */
/* Genericity is in principle doable, but too             */
/* complex for this demonstrator                          */
```

```
/* singleton interface class */

#include <map>
using namespace std;

class MtxIntf {
public:
  static MtxIntf* getIntf();

  long create(int nr, int nc);
  long read(string name);
  void write(long ref, string name);
  void destroy(long ref);

  Mtx* getMtx(long ref); // to be called by plain C
                         // functions from fortran interface

private:
  MtxIntf() {}
  MtxIntf(MtxIntf const&) {abort();}

  MtxStore store;
  map<long, Mtx*> mtxMap;
  static MtxIntf* theIntf;
  static long lastRef;
};

/* And now the functions that can be called from fortran */

extern "C" {
  long mtxcreate_(int* nr, int* nc);
  long mtxread_(char* nm, int len);
  void mtxwrite_(long* ref, char* nm, int len);
  void mtxdestroy_(long* ref);

  void mtxgetdim_(long* ref, long* nr, long* nc);
  void mtxinverse_(long* ref);
  void mtxgetdata_(long* ref, float* data);
  void mtxsetdata_(long* ref, float* data);
}

#endif
```

The file `mtxintf.cc`

```
#include "mtxstore.h"
#include "mtxintf.h"

/* Fortran interface to planck matrix objects */
/* For demonstration only                     */
/* Eric Aubourg, CEA/DAPNIA/SPP May 2000      */
```

```
/* Requirements: we want to instanciate objects    */
/* from a main fortran code, call standard planck  */
/* methods on those objects, and access data from  */
/* fortran                                         */
/* We will access objects from Fortran using       */
/* integer values, in a way similar to logical     */
/* units, since there is no pointer in Fortran     */
/* The C code needs to keep a map of int -> object */

/* A similar interface must be written for every   */
/* object, unless we attempt genericity            */
/* Genericity is in principle doable, but too      */
/* complex for this demonstrator                   */

MtxIntf* MtxIntf::theIntf = 0;
long     MtxIntf::lastRef = 0;

/* singleton provider */
MtxIntf* MtxIntf::getIntf()
{
 if (!theIntf) theIntf = new MtxIntf();
 return theIntf;
}

long MtxIntf::create(int nr, int nc) {
  lastRef++;
  mtxMap[lastRef] = new Mtx(nr,nc);
  return lastRef;
}

long MtxIntf::read(string name) {
  lastRef++;
  mtxMap[lastRef] = new Mtx(store.fetch(name));
  return lastRef;
}

void MtxIntf::write(long ref, string name) {
  store.store(*mtxMap[ref], name);
  // Not handled : check validity of ref
}

void MtxIntf::destroy(long ref) {
  delete(mtxMap[ref]);// Not handled : check validity of ref
  mtxMap.erase(ref);
}

Mtx* MtxIntf::getMtx(long ref) {
  // to be called by plain C functions from fortran interface
  return mtxMap[ref];
```

```
}

/* And now the functions that can be called from fortran */

long mtxcreate_(int* nr, int* nc) {
  return MtxIntf::getIntf()->create(*nr, *nc);
}

long mtxread_(char* nm, int len) {
  return MtxIntf::getIntf()->read(string(nm,len));
}

void mtxwrite_(long* ref, char* nm, int len) {
  MtxIntf::getIntf()->write(*ref,string(nm,len));
}

void mtxdestroy_(long* ref) {
  MtxIntf::getIntf()->destroy(*ref);
}

void mtxgetdim_(long* ref, long* nr, long* nc) {
  Mtx* m = MtxIntf::getIntf()->getMtx(*ref);
  *nr = m->getNRows();
  *nc = m->getNCols();
}

void mtxinverse_(long* ref) {
  MtxIntf::getIntf()->getMtx(*ref)->inverse();
}

void mtxgetdata_(long* ref, float* data) {
  MtxIntf::getIntf()->getMtx(*ref)->copyToForArray(data);
}

void mtxsetdata_(long* ref, float* data) {
  MtxIntf::getIntf()->getMtx(*ref)->copyFromForArray(data);
}
```

Fortran/C++ interfacing, which way ?

From the previous examples, it is quite clear that manipulating C++ objects from Fortran is possible, but needs a fair amount of work from DPC and DMCI implementors. Linking the final program can be tricky and platform-dependant.
Considering the minimum amount of learning (basically copy-pasting examples) required by the other solution (a main C++ program calling Fortran procedures), it should probably be preferred.

Calling Fortran from Java

[in progress] The Java Native Interface is described in the DMS document *Going native seamlessly - calling C from Java*, and we will use it to create a C bridge between Java and Fortran.
A Java class must be created (which can be the class containing the "main" function) with a method declared `native`. Running the `javah` program on the java source will create C headers. The native method can then be implemented in C.
To get access to Java objects — call methods, or read/write attributes — from C, one must use the JNI API. Java objects are identified by *interface pointers*, and methods and attributes by their name (including signature for methods).
The C implementation of our computation method can then extract the data of the matrix — a copy might be done or not, depending on the implementation, since the physical layout of objects in memory is not specified in the Java Virtual Machine specifications to allow optimization. It can then call the Fortran code, and store back the matrix data.
If data must be kept from one call to the other, care must be taken to interact properly with the Java garbage collector.
The C and Fortran code must then be compiled and linked as a shared library (.so file on Unix). The shared library must be visible by the `java` executable (LD_LIBRARY_PATH on Unix), and loaded by the Java program.
[code exemple to be inserted]

Calling Java from Fortran

The trickiest part for the end…
From C, the JNI API provides all the functions to generically manipulate Java object : a Java Virtual Machine must be created from the C program, and then objects can be created, methods can be called, etc.
From Fortran, the easiest way will be to have a 64-bit INTEGER (i.e. long enough to represent a pointer) have the value of the JNI interface pointer. This number will thus be our reference to Java objects. JNI will handle the type-checking and provide some security and debugging help. Since genericity is present in JNI (classes, methods, attributes are designated by their name through a string), a Fortran interface to JNI can keep this genericity. For our simple demonstrator, we will stick with our non-generic model.
[to be completed]

Annex A: the `Mtx` class, C++ version (provided by DPC)

File `mtx.h`

This file describes the interface for the `Mtx` class. Functions are provided to copy to data to and from a Fortran array, handling the difference of storage (row-major or column-major) between C and Fortran multi-dimensional arrays. Note that `SOPHYA` arrays can be configured to have a Fortran layout, sparing the copy — this won't work for Java, where array layouts are not defined in the JVM specifications for optimization purposes : the access to array data might do a copy if necessary even for monodimensional arrays, in a system-dependant way.

```
/* A matrix class for demonstration only */
/* Eric Aubourg, CEA/DAPNIA/SPP May 2000 */

/* This class would be provided by the standard */
/* Planck DPC OO framework */
```

```
#ifndef MTX_H
#define MTX_H

class Mtx {
public:
  Mtx(int nr, int nc);
  ~Mtx();
  Mtx(Mtx const&);
  Mtx& operator = (Mtx const&);
  float& operator() (int i, int j) {return data[i*ncols+j];}
  float operator() (int i, int j) const {return data[i*ncols+j];}
  void inverse();
  void copyToForArray(float*) const;
  void copyFromForArray(float const*);
  int getNCols() const {return ncols;}
  int getNRows() const {return nrows;}
protected:
  int nrows;
  int ncols;
  float* data;
};

#endif
```

The file `mtx.cc`

A C++ dummy implementation.

```
/* A matrix class for demonstration only */
/* Eric Aubourg, CEA/DAPNIA/SPP May 2000 */

/* This class would be provided by the standard */
/* Planck DPC OO framework */


#include "mtx.h"
#include <string.h>

Mtx::Mtx(int nr, int nc)
  : nrows(nr), ncols(nc), data(new float[nr*nc])
{}

Mtx::~Mtx()
{
  delete[] data;
}

Mtx::Mtx(Mtx const& a)
  : nrows(a.nrows), ncols(a.ncols), data(new float[nrows*ncols])
```

```
{
  memcpy(data, a.data, nrows*ncols*sizeof(float));
}

Mtx& Mtx::operator = (Mtx const& a)
{
  delete[] data;
  nrows = a.nrows;
  ncols = a.ncols;
  data = new float(nrows*ncols);
  memcpy(data, a.data, nrows*ncols*sizeof(float));
  return *this;
}

void Mtx::inverse()
{
  //super smart optimized multi processor matrix inversion
  for (int j=0; j<ncols; j++)
    for (int i=0; i<nrows; i++)
      (*this)(i,j) += i+j;
}

void Mtx::copyToForArray(float* a) const {
  for (int j=0; j<ncols; j++)
    for (int i=0; i<nrows; i++)
      *(a++) = (*this)(i,j);
}

void Mtx::copyFromForArray(float const* a) {
  for (int j=0; j<ncols; j++)
    for (int i=0; i<nrows; i++)
       (*this)(i,j) = *(a++);
}
```

Annex B: the `MtxStore` class, C++ version (provided by DMCI)

File `mtxstore.h`

```
/* A matrix persistence class for demonstration only */
/* Eric Aubourg, CEA/DAPNIA/SPP May 2000             */

/* This class would be provided by the standard */
/* Planck DMCI */

#ifndef MTXSTORE_H
#define MTXSTORE_H

#include <string>
#include "mtx.h"
```

```
using namespace std;

class MtxStore {
public:
  Mtx fetch(string);
  void store(Mtx const&, string);
};

#endif
```

File `mtxstore.cc`

```
/* A matrix persistence class for demonstration only */
/* Eric Aubourg, CEA/DAPNIA/SPP May 2000             */

/* This class would be provided by the standard */
/* Planck DMCI */

#include "mtxstore.h"
#include <iostream>
#include <fstream>

Mtx MtxStore::fetch(string n) {
  cout << "reading " << n << endl;
  Mtx m(10,10);
  for (int i=0; i<10; i++)
    for (int j=0; j<10; j++)
      m(i,j) = j*i+i+2*j+3;
  return m;
}

void MtxStore::store(Mtx const& m, string n) {
  cout << "storing " << n << endl;
  ofstream f(n.c_str());
  for (int i=0; i<m.getNRows(); i++) {
    for (int j=0; j<m.getNCols(); j++)
      f << m(i,j) << " ";
    f << '\n';
  }
}
```