

Earth System Modeling Framework
Implementation Report

David Neckels and Cecelia DeLuca

May 4, 2007

Contents

1	Introduction	2
2	Motivation and Assessment Strategy	2
3	Shallow and Deep Classes	2
3.1	C++ Implementation	3
3.2	F90 Implementation	6
4	Inheritance	9
4.1	C++ Implementation	9
4.2	F90 Implementation	12
4.3	C Implementation	15
5	Templates	15
5.1	C++ Implementation	15
5.2	F90 Implementation	19
5.3	C Implementation	19
6	Array Sharing	19
6.1	Memory Ordering	20
6.2	Array Sections	21
7	Pointer Size	24
7.1	C/C++ Implementation	24
8	Function Pointers	25
8.1	Basic C function Pointers	25
9	Optional Arguments in C++	27
10	Performance	28
10.1	Single-Language Computation	28
10.2	Large-Scale Example	28
11	Linking C++ and F90	29
11.1	SunOS	29
11.2	AIX	29
11.3	IRIX	29
11.4	Linux	30
12	Using Shared Libraries	30
13	Tools	30
13.1	Chasm	30
13.2	Babel	30
14	Conclusions	31
15	Glossary	31

1 Introduction

The Earth System Modeling Framework (ESMF) effort is developing software *infrastructure* for building high-performance Earth system model components, and software *superstructure* for assembling those components into applications. [?] The quality of the ESMF implementation will be a determining factor in its success, and the implementation issues it faces are nontrivial. The ESMF implementation must address two key questions related to computer language:

- What language or languages will the framework be implemented in? and
- What is the strategy for language interoperability?

In this *Implementation Report* we present a series of exercises and examples that guided us in formulating answers to these questions. Our conclusions are presented in the final section of the document.

The code examples in this *Report* may be downloaded from the ESMF SourceForge repository at <http://www.sourceforge.net/projects/esmf>. Select the **Code** item on the main menu bar, and **CVS Browse** from its pull-down menu. Instructions for download are on that page. Download the module `impl_rep`. Note that the ESMF team no longer supports or tests this code!

2 Motivation and Assessment Strategy

The ESMF must provide an F90 interface and, less critically in the short term, a C++ interface. [?] An immediate question arises as to whether the ESMF should be *implemented* in F90 or whether it may be implemented in some other language and *wrapped* in F90. The wrapping approach is certainly not new, as many systems libraries (MPI [?], netCDF [?], etc..) with Fortran interfaces are programmed in C. The strategy selected must offer excellent performance, robustness, concise and maintainable code, and interfaces natural for the application programmer. It is also useful for both library and application programmers facing complex software tasks to be able to use a language rich in data structures and object-oriented (OO) constructs.

We consider C, C++ and F90 all viable implementation language candidates, as each has a significant base of developers and all currently support high-performance applications. C offers excellent portability and performance but has weaker support for object-oriented features, such as polymorphism, that F90 and C++ can offer. Interfacing C to F90 and to C++ is routine but can be ponderous. Interfacing F90 and C++ directly, with either as the implementation language, is relatively uncommon and no standard, mature approach exists. There are a number of efforts currently underway intended to address this gap. [?, ?] The ESMF team maintains a productive collaboration with Chasm, perhaps the most mature tool available for automatic generation of C++/F90 interfaces. The potential for using Chasm in ESMF is discussed in Section 13.1.

In order to evaluate implementation and interoperability strategies we have prepared a series of examples. All are implemented in C++ and F90, and in each case we obtained C++ and F90 interfaces. C implementations are presented or discussed where they offer special insights - since C-based strategies are relatively well-understood, we have not focused on them. In the examples we have tried to retain a programming style natural for the calling language. For example, we have tried not to impose a strong C++ accent on F90 when it wraps C++ code.

The examples illustrate the aspects of implementation language and language interoperability that we believe are relevant for the ESMF. We begin with a discussion of *shallow* and *deep* classes, two different strategies for handling the inter-language interface. We next explore language features such as inheritance and templating, looking at both the capabilities of implementation languages and the degree to which these OO concepts can carry across the language boundary. We look carefully at performance, using both single-processor examples that quantify implementation language capabilities, and a pertinent large-scale example that includes both communication and the overhead costs of language wrapping. Finally we examine some of the details of language interoperability, such as handling pointer size and storage order.

We have run all examples on AIX, IRIX, Sun and Linux compilers.

3 Shallow and Deep Classes

We will use the term “classes” generically to refer not only to class structures in C++ but also to derived types in F90 and structs in C. Two distinct schemes were developed for handling classes that will traverse language boundaries. The distinction is based on whether memory allocation for the class occurs on the calling language side or on the implementation language side. *Shallow* classes do not have any memory allocated on the implementation language side - memory is passed to them from the calling language. Their memory can be drawn from the stack in the local procedure, so there is no need for the calling language to “create” and “destroy” them. *Deep* classes allocate memory on the implementation language side, and therefore must be “created” and “destroyed” by the calling language.

Shallow classes have all function calls dispatched at compile time, whereas deep classes implemented in C++ may actually have virtual function tables and runtime dispatching of calls. Either shallow or deep classes may work on or may contain pointers to arrays that are allocated by the calling language. If this is the case, the expectation is that procedures in the implementation language will not be responsible for deallocating or altering the memory associated with the “foreign language” array.

All classes are assumed to be “opaque” from the calling language side; that is, accessor functions are required to retrieve data or attributes embedded in a class. *All data members in our examples are accessed from the implementation language side.* This restriction can be lifted in the case of FORTRAN calling C with the introduction of the F200X BIND(C) attribute for derived types [?]. We note that for the compilers we have used simple F90 derived types are memory identical to C structs or C++ classes, and it would be possible to directly access class data using either the calling or implementation language. However, it does not appear necessary or desirable at this point to utilize this capability since it may degrade the robustness of the implementation considerably in unpredictable ways.

The shallow class approach is warranted for very simple classes within the ESMF. A `Date` class, for example, does not need to contain internal pointers and would, if it was created within a single-language application, normally be handled as a local stack variable. Deep classes are needed to represent more complex classes within the framework, such as a `Route` class that stores data transfer paths between two components. Such classes are likely to allocate storage within the construction or other methods of the class.

3.1 C++ Implementation

Code for the example is in `lightweight_example/C++`.

Shallow Object Example

The following example illustrates shallow objects with a particle that has an associated position and velocity. The example is an adaption of the particle object in [?].

In C++:

```
#include <iostream.h>
#include <ftn.h>
#include <stdio.h>
class particle {
public:
    virtual ~particle() = 0;
    particle_init(double *position_, double *velocity_) {
        int i;
        for (i = 0; i < 3; i++) {
            position[i] = position_[i];
            velocity[i] = velocity_[i];
        }
    }
    void move(double &timeStep) {
        position[0] += velocity[0] * timeStep;
        position[1] += velocity[1] * timeStep;
    }
};
```

```

    position[2] += velocity[2] * timeStep;
}
void print() {
    int i;
    for (i=0; i < 3; i++)
        cout << "position[" << i << "] = " << position[i] << endl;
}
private:
    double position[3], velocity[3];
};

```

This section of code defines the `particle` class. Functions for integrating movement and printing out `particle` position are included in the example.

This C++ code doubles as the implementation *and* the C++ interface. However, to use this class in F90 we must build an analogous F90 module:

```

module particleMod

type particle
sequence
real(r8) position(3)      ! These variables exist to create a struct
real(r8) velocity(3)     ! with the same size as the corresponding
end type                  ! C++ class. They are not used directly.

contains

subroutine particleCreate(this, position, velocity)
type(particle), intent(inout) :: this
real(r8), intent(in) :: position(:), velocity(:)
call c_particle_create(this, position, velocity)
end subroutine

subroutine particleMove(this, timeStep)
type(particle) :: this
real(r8) :: timeStep

call c_particle_move(this, timeStep)

end subroutine

subroutine particlePrint(this)
type(particle) :: this

call c_particle_print(this)

end subroutine

end module

```

This interface contains a datatype that is the same size as the `particle` class. This assures that when the object is created in F90, it will have the correct amount of memory available to be used by the C++ class. F90 interface

functions/subroutines are included here to call the appropriate C++ class member functions. These F90 routines do not perform any real work (the implementation is in C++), but they map the F90 call to the C++ member function. An intermediate level of C code is required since C++ name-mangles its functions (the macro FTN is a cpp macro that maps a C function to the appropriate F77 linkage; i.e. it adds underscores as needed):

```
// Interface Layer for F90
extern "C"
{
    void FTN(c_particle_create)(particle *pa, double *position, double *velocity) {
        pa->particle_init(position, velocity);
    }
    void FTN(c_particle_move)(particle *pa, double *timeStep) {
        pa->move(*timeStep);
    }
    void FTN(c_particle_print)(particle *pa) {
        pa->print();
    }
}
```

The extern "C" in the interface tells the C++ compiler to give the interface functions C linkage, so they are immediately callable from Fortran.

When a section of code defines a particle on the F90 stack, the compiler allocates space for it and the C++ routines use that memory to operate on. The F90 structure address is passed to the C interface function, where the type of the address space is cast to a type particle*.

```
void FTN(c_particle_move)(particle *pa, double *timeStep) {
    pa->move(*timeStep);
}
```

The resulting address is used to call the member function. This function lookup actually occurs at compile time, and the pointer is merely sent in as the this pointer.

With this simple class, there are no virtual function tables or complex structures associated with the class. Hence, a function lookups are automatic. Consider:

```
#include <iostream.h>
class example {
public:
    void print()
    {
        cout << "hello world\n";
    }
};

main()
{
    ((example*)(NULL))->print();
}
```

This program works and prints "hello world". Thus when we pass our chunk of memory allocated from F90, the correct class member function is dispatched. In the compilers we have worked with, C++ *doesn't store any any information about how to find functions within the class memory itself (for shallow classes)*. All function lookups in

these classes are done at compile time and the image of the C++ class in memory is exactly like a C struct with the same data members. Once inheritance and more complicated techniques are involved, this does not work and we must move to our second “deep” scheme, discussed later. Our F90 main would look like this:

```
program particleEX
Use particleMod

call useparticle()

contains

subroutine useparticle()
type(particle) :: mypart
real(r8) :: position(3) = (/ 1.01, 1.0, 1.0 /)
real(r8) :: velocity(3) = (/ 0.5, 0.0, 0.1 /)
integer :: i
real(r8) :: timeStep

call particleCreate(mypart, position, velocity)
timeStep = 1.0

do i=1, 10
    call particleMove(mypart, timeStep)
enddo
call particlePrint(mypart)

end subroutine

end program
```

The major convenience of this strategy is that we have allocated and deallocated memory for our particle on the F90 stack *automatically*. The user has the convenience of not having to explicitly call a `destroy` routine.

3.2 F90 Implementation

Code for the example is in `lightweight_example/F90`.

Shallow Object Example

We again use the particle example, implemented in F90 this time, and wrapped in C++.

```
module particleMod

type particle
sequence
real(r8) position(3)
real(r8) velocity(3)
real(r8) mass
end type

contains
```

```

subroutine particleCreate(this, position, velocity)
type(particle), intent(inout) :: this
real(r8), intent(in) :: position(:), velocity(:)
integer :: i
do i=1, 3
    this%position(i) = position(i)
    this%velocity(i) = velocity(i)
enddo
end subroutine

```

```

subroutine particleMove(this, timeStep)
type(particle) :: this
real(r8) :: timeStep
integer :: i
do i=1, 3
    this%position(i) = this%position(i) +
        &    this%velocity(i) * timeStep
enddo

```

Next, the module must be made callable by C/C++ code. It is not feasible to portably link directly to module functions, since module routines have unpredictable linkage, and since F90 in general requires call headers to be on the stack when a function is called. Therefore we had to go through F77 functions, which are simpler both in linkage and calling. This way the module functions are called from FORTRAN code, and the language deals with the linkage and call stack issues.

```

subroutine f_particlecreate(this, position, velocity)
Use particleMod
type(particle) :: this
real(r8) :: position(3)
real(r8) :: velocity(*)
call particleCreate(this, position, velocity(1:3))
end

```

```

subroutine f_particleprint(this)
Use particleMod
type(particle) :: this
call particlePrint(this)
end

```

```

subroutine f_particlemove(this, timeStep)
Use particleMod
type(particle) :: this
real(r8) :: timeStep
call particleMove(this, timeStep)
end

```

These functions are now callable from C/C++ on the compilers we have tried.

The array parameters `position` and `velocity` are not used as assumed shape parameters (i.e. `real(r8) :: position(:)`) because this requires that a call header be placed on the stack describing the dimensional

characteristics of the array. The format of this *call header* is not public or standardized between different compilers, so it is difficult to use. (See Section 6.2 for strategies for working with Fortran arrays.) This example employs the two simpler methods of array passing that require no call headers, namely the explicit-shape method (`position(3)`) and the old F77 array passing method (`velocity(*)`).

Now we have functions that we can call from C/C++, so we create a C++ class that maps member functions to these F90 interfaces (via C):

```
class particle;

extern "C" {
    // These prototype the F77 interface functions.
    void FTN(f_particlecreate)(particle *pa, double *position,
        double *velocity);
    void FTN(f_particlemove)(particle *pa, double *timeStep);
    void FTN(f_particleprint)(particle *pa);
}

class particle {
public:
    particle(double *position_, double *velocity_) {
        FTN(f_particlecreate)(this, position_, velocity_);
    }
    void move(double &timeStep) {
        FTN(f_particlemove)(this, &timeStep);
    }
    void print() {
        FTN(f_particleprint)(this);
    }
private:
    double position[3], velocity[3];
    double mass;
};
```

With this class definition we have essentially turned the C++ shallow class example inside out, and now the C++ member functions are merely stubs to call F90 code, as the F90 module procedures in our earlier example were stubs to call C++ code. In these shallow classes we still have corresponding data structures on both sides of the language interface.

Our C++ driver uses this class as if it were any ordinary C++ class:

```
int main(int argc, char *argv[]) {
    int i;
    double timestep;
    double position[] = { 1.1, 1.2, 1.3};
    double velocity[] = { 0.5, 0.0, 0.1};
    particle part1(position,
velocity);
    timestep = 1.0;
    for (i = 0; i < 10; i++) {
        part1.move(timestep);
    }
}
```

```

    }
    part1.print();
}

```

Deep Object Example

The other language interoperability strategy involves objects of more complexity, such as classes with inheritance, or classes that construct and destruct resources. These *deep* classes must have legitimate constructors and destructors and we expect the memory created/initialized by the constructors to be referenced when using the class. *Deep* classes may be associated with data related to OO constructs such as virtual function tables, or may allocate memory during construction or operations. Examples of *deep* objects are provided in the Sections 4 and 5.

4 Inheritance

Inheritance is the ability to create a *derived* class that includes the properties of one or more other *superclasses* or *base* classes. One of the most common uses of inheritance is to define a set of different specialized classes from a more general class.

There are two important benefits to inheritance, the first being the ability for a derived class to utilize the functionality of the base class, the second being polymorphic behavior, or the ability for a base class to be used as an interface for all derived classes.

This first benefit of inheritance can be emulated in C and F90 simply through class containment. In the derived class we include an object of the base type and provide an interface by which the user of the object may call those functions. This emulation requires more typing and extra constructs in C and F90 than C++ but the functionality is entirely possible.

The second benefit, polymorphism, is the ability for a class to take on different behavior depending on its type. A typical example is the `window->drawPixel` routine, which allows an algorithm to create a window object and draw pixels without worrying what graphics library is beneath the hood (X11, OpenGL, Win32, etc...). To implement this behavior you must have the concept of a base class pointer, which can point to any class in the inheritance hierarchy. The base class defines virtual functions which act as an interface for all derived types of the class. When a virtual function is called from a base pointer, the runtime environment figures out what the true type of the object is and dispatches the call to the appropriate derived function. This is an extremely powerful capability that allows us to write complex algorithms using a base object which operates on a whole set of derived types.

Polymorphic behavior is often implemented in C using the `void*` and an enumerated class-type field. Since C supports function pointers, a virtual function table may be constructed, if desired. F90, on the other hand, does not allow this kind of “under the hood” manipulation that the `void*` provides, and makes it far more difficult to implement polymorphism. The biggest stumbling block is that a base pointer is not really possible (we lack a casting operator), and we end up implementing the base class as something more like a union of derived classes. This is developed further in the example.

This implementation of inheritance in F90 is developed at length in [?].

F90 does, however, offer a means of supporting polymorphism through the creation of generic functions. A generic procedure (that is, a shared procedure name) can be created for an assortment of specific procedures through the `interface` keyword. The specific procedure is resolved by matching the argument list in the call to the arguments of the procedures that share the interface. Thus an interface that appears to support polymorphism through inheritance can be created using F90. This straightforward implementation of polymorphism in F90 does not perform run-time dispatching. However, run-time polymorphism can be supported in F90 through the creation of a polymorphic class (see [?]).

4.1 C++ Implementation

Code for the example is in `inheritance_example/C++`.

This example illustrates how an F90 interface can be affixed to a C++ inheritance hierarchy.

The classes:

```
#include <iostream.h>
#include "ftn.h"
class climber {
public:
    virtual ~climber() {}
    void speak() {
        cout << "I climb ";
        speak_specific();
    }
    virtual void speak_specific() {}
};
class alpine : public climber {
public:
    void speak_specific() {
        cout << " snowy couloirs" << endl;
    }
};
class rock : public climber {
public:
    void speak_specific() {
        cout << " rock faces" << endl;
    }
};
```

The F90 interface module:

```
module climberMod

type climberPtr
sequence
integer(r8) this
end type

contains

function alpineCreate()
type(climberPtr) :: alpineCreate
call c_alpine_create(alpineCreate)
end function

function rockCreate()
type(climberPtr) :: rockCreate
call c_rock_create(rockCreate)
end function

subroutine climberSpeak(this)
type(climberPtr), intent(in) :: this
call c_climber_speak(this)
end subroutine
```

```

subroutine climberDestroy(this)
type(climberPtr), intent(in) :: this
call c_climber_destroy(this)
end subroutine

end

```

The F90 `type` definition in this example differs significantly from that in the shallow example in that the C++ and F90 types are no longer associated with similar structures in memory. The F90 `type` in this example is now merely a handle with enough memory to store the C++ `this` pointer.

The other significant difference is that the memory now is allocated dynamically on the C++ side (C++ must be allowed to populate the virtual function tables), not the F90 side.

The C “glue” code (same as lightweight example except for the inclusion of an allocation and deallocation routine and an additional layer of pointer indirection):

```

extern "C" {

    void FTN(c_alpine_create)(climber **ths) {
        climber *ap = new alpine();
        *ths = ap;
    }
    void FTN(c_rock_create)(climber **ths) {
        climber *ap = new rock();
        *ths = ap;
    }
    void FTN(c_climber_speak)(climber **ap) {
        (*ap)->speak();
    }
    void FTN(c_climber_destroy)(climber **ap) {
        delete (*ap);
    }
}

```

An example F90 driver:

```

program climberEx
Use climberMod
type(climberPtr) :: climber1, climber2

climber1 = alpineCreate()
climber2 = rockCreate()
call climberSpeak(climber1)
call climberSpeak(climber2)
call climberDestroy(climber1)
call climberDestroy(climber2)

```

In this example the F90 driver creates several different subclasses of `climber`. The F90 program is able to use an identical interface to each class (the one provided via virtual functions in C++). Hence, F90 gets all of the benefits of the C++ inheritance hierarchy. This example actually extends the capabilities of F90.

The F90 program stores a pointer to the class, and within the memory image of that class is all the information about what type the class is and how to dispatch the correct virtual functions for that class.

The C interface functions accept the address of the F90 type, in which is stored the address of a class. Hence, the C function receives a pointer to pointer:

```
void FTN(c_climber_speak)(climber **ap)
{
  (*ap)-> speak();
}
```

4.2 F90 Implementation

Code for the example is in `inheritance_example/F90`.

Implementing an inheritance scheme is possible in F90, though it is cumbersome. This example demonstrates one of the methods that might be used to implement inheritance if F90. It is one of the models that Szymanski and Decyk [?] use:

```
module climberMod
integer, parameter :: ALPINE = 1,
    & ROCK = 2

    type alpinePtr
sequence
integer :: some_alpine_data
end type

    type rockPtr
sequence
integer :: some_rock_data
end type

type climberPtr
sequence
real(8) :: climber_data
integer :: type
type(alpinePtr), POINTER :: alpine
type(rockPtr), POINTER :: rock
end type

contains

subroutine alpineInit(alpineCreate)
type(climberPtr) :: alpineCreate
allocate(alpineCreate%alpine)
alpineCreate%type = ALPINE
end subroutine

subroutine rockInit(rockCreate)
type(climberPtr) :: rockCreate
allocate(rockCreate%rock)
rockCreate%type = ROCK
end subroutine

subroutine climberSpeak(this)
```

```

type(climberPtr) :: this
print *, "I climb "
if (this%type .eq. ALPINE) then
  call alpineSpeak(this)
else if (this%type .eq. ROCK) then
  call rockSpeak(this)
endif

end subroutine

subroutine alpineSpeak(this)
type(climberPtr), intent(in) :: this
print *, " snow couliours"
end subroutine

subroutine rockSpeak(this)
type(climberPtr), intent(in) :: this
print *, " rock faces"
end subroutine

subroutine climberDestroy(this)
type(climberPtr), intent(in) :: this
if (this%type .eq. ALPINE) then
  deallocate(this%alpine)
else if (this%type .eq. ROCK) then
  deallocate(this%rock)
endif
end subroutine

end

```

The example uses a *union* approach, where the object contains pointers to all the possible derived types. Only one of these pointers is allocated, and a type field is used to keep track of the derived class type. In virtual functions (`climberSpeak`), this type member is used as a switch to the correct function. This solution is entirely functional, although it wastes some space (the derived pointers), and it is more of a *tabled* approach, since adding a new class requires modifying the base object. This violates *extensibility* at least at the source level, though with care the programmer will not change any of the base class interfaces.

Wrapping the F90 module in C++ is similar to the reverse wrapping, however an extra layer of indirection (through F77) is necessary.

The F90 interface code:

```

subroutine f_alpinecreate(this_arg)
Use climberMod
integer :: this_arg
type(climberPtr), POINTER :: this
allocate(this)
call alpineInit( this )
call ESMF_loc(this_arg, this)
end subroutine

subroutine f_rockcreate(thisa)
Use climberMod

```

```

integer this_arg
type(climberPtr), POINTER :: this
allocate(this)
call rockInit(this)
call ESMF_loc(this_arg, this)
end subroutine

subroutine f_climberspeak(this)
Use climberMod
type(climberPtr) :: this
call climberSpeak(this)
end subroutine

```

We have created our own version of loc, since loc does not always return the address of the newly created type from the POINTER. Its behavior appears to vary from platform to platform.

Our version of loc, ESMF_loc:

```

void FTN(esmf_loc) (void ** return_arg, void *pointer_arg)
{
    *pointer_arg = return_arg;
}

```

The C++ wrapper class is below:

```

class climber;

extern "C" {
    void FTN(f_climberspeak)(climber *);
    void FTN(f_alpinecreate)(climber **);
    void FTN(f_rockcreate)(climber **);
}

class climber {
public:
    virtual ~climber() {}
    virtual void speak() {
        FTN(f_climberspeak)(handle);
    }
protected:
    climber *handle;
    int variable;
};

class alpine : public climber {
public:
    alpine() {
        FTN(f_alpinecreate>(&handle);
    }
};

```

```

class rock : public climber {
public:
    rock() {
        FTN(f_rockcreate)(&handle);
    }
};

```

This code stores the address of the FORTRAN type in the class variable `handle`. It passes this address back to the F90 functions. This is the same basic approach that was used in the opposite direction.

4.3 C Implementation

Inheritance from a superclass can be implemented in C by having the superclass struct be the first entry in a derived class struct (see, e.g. Rumbaugh). Thus all the attributes and operations of the superclass are included in the derived class. Since the superclass methods begin accessing the data objects they receive based on the pointer to the start of the struct, and they have no knowledge of memory in the struct beyond this, derived objects can be input into methods of either the superclass or the derived class.

Though this method cannot support multiple inheritance, it is an efficient way of reusing the same implementation of an operation for multiple classes. For example, a `Vector` class and a `Matrix` class that inherits from the `Vector` class can both use the same set of elementwise `Vector` operations. Typically an elementwise `Matrix` operation would be created corresponding to each elementwise `Vector` operation and assigned the function pointer from the `Vector` operation. Thus a programmer might pass a `Matrix` object into a `Matrix_Clear` method and not be aware that the operation was actually a `Vector_Clear` performed on the `Vector` embedded in the `Matrix` object.

5 Templates

Templating is a way of creating generic functions that perform the same operations on multiple kinds of data. Ideally the programmer only has to write the implementation once, and the language can determine or enable the programmer to specify the particular kind of data desired.

5.1 C++ Implementation

Code for the example is in `template_example/C++`.

C++ has built-in templating that enables the programmer to define an operation based on one or more generic types. The type is only required to define the operations that are used in the templated procedure.

It was originally feared that wrapping C++ classes in F90 would seriously limit the ability to use templates, but upon investigation we found that C++ templates may be used to whatever extent desired. The slight inconvenience encountered is that for all desired types the templated procedure must be known in advance and coded into the interface. Still, the same implementation code can be used for multiple template types: only the *interface* must be duplicated. This is a step beyond what an implementation in F90 would grant; in other languages the code as well as the interface would need to be duplicated for each desired type.

As an example to demonstrate both the power of templating and the method for wrapping templates in F90, the following example builds a generic polynomial class with some basic operations. The example is short and only provides a few operations for the sake of clarity. The reader can imagine extending this class to provide many more operations such as root finding, derivatives, etc...

The extraordinary thing about this example is that the code is written for a generic algebraic type (`DTYPE`). When this is actually used the user must specify the algebraic type. The only restriction on the type is that it implement the operations used within the algorithm, namely “+/*”. This type could be `double`, `complex`, `int` or some user defined type.

```

#include<iostream.h>
#include <math.h>
#include <complex.h>
#include "ftn.h"
template<class DTYPE>
class poly {
public:
    poly(int order_, DTYPE *coefficients_) {
        int i;
        order = order_;
        coefficients = new DTYPE[order + 1];
        for (i = 0; i < order + 1; i++)
            coefficients[i] = coefficients_[i];
    }
    ~poly() {
        delete [] coefficients;
    }
    void evaluate(DTYPE &x, DTYPE &res) {
        int i, j;

        // res = (DTYPE) 0; (this would require DTYPE to implement assignment from int).
        res -= res;
        for (i=0; i <= order; i++) {
            res *= x;
            res += coefficients[i];
        }
    }
    void get_order(int &order_) {
        order_ = order;
    }
    // void findRoots(DTYPE *roots);
    // etc...
private:
    DTYPE *coefficients;
    int order;
};

```

An F90 module interface:

```

module polyMod

type polyPtr
private
sequence
integer(r8) :: this
end type

interface polyCreate
module procedure polyCreateReal
module procedure polyCreateInt
end interface

```

```

interface polyEvaluate
module procedure polyEvaluateReal
module procedure polyEvaluateInt
end interface

contains

function polyCreateReal(order, coefficients)
integer, intent(in) :: order
real(r8) :: coefficients(:)
type(polyPtr) :: polyCreateReal
call c_poly_create_double(polyCreateReal, order, coefficients)
end function

function polyCreateInt(order, coefficients)
integer, intent(in) :: order
integer :: coefficients(:)
type(polyPtr) :: polyCreateInt
call c_poly_create_int(polyCreateInt, order, coefficients)
end function

subroutine polyGetOrder(this, order)
type(polyPtr) :: this
integer, intent(out) :: order
call c_poly_get_order(this, order)
end subroutine

subroutine polyEvaluateReal(this, x, res)
type(polyPtr) :: this
real(r8) :: x
real(r8), intent(out) :: res
call c_poly_evaluate_double(this, x, res)
end subroutine

subroutine polyEvaluateInt(this, x, res)
type(polyPtr) :: this
integer :: x
integer, intent(out) :: res
call c_poly_evaluate_int(this, x, res)
end subroutine

subroutine polyDestroy(this)
type(polyPtr) :: this
call c_poly_destroy(this)
end subroutine

end module

```

There are significant similarities between this module interface and the previous inheritance example. The F90 derived type defines a handle, and each function maps to one of the C++ member functions.

There are some new things as well. In this example the two template parameters implemented are Int and Real.

A separate call for each routine must be generated based on the type. The F90 interface feature nicely maps these back to a single function for the module user.

The C “glue” code is now a bit more complex. Since writing a function that uses a `poly<DTYPE>` pointer is no longer a real function but just a function template, a separate call must be made to the template function for each type that the interface supports:

```
// Generic interface functions
template<class DTYPE>
void c_poly_create(poly<DTYPE> **ths, int *order, DTYPE *coefficients) {
    *ths = new poly<DTYPE>(*order, coefficients);
}
template<class DTYPE>
void c_poly_evaluate(poly<DTYPE> **ths, DTYPE *x, DTYPE *res) {
    (*ths)->evaluate(*x, *res);
}

extern "C" {
    // Order doesn't care about template type. Use int as placeholder. Could
    // use void*, but this class requires arithmetic operators.
    void FTN(c_poly_get_order)(poly<int> **ths, int *order) {
        (*ths)->get_order(*order);
    }
    void FTN(c_poly_destroy)(poly<int> **ths) {
        delete *ths;
    }
    // Now implement for double and int. Could be any user defined class.
    void FTN(c_poly_create_double)(poly<double> **ths,
int *order, double *coefficients) {
        c_poly_create(ths, order, coefficients);
    }
    void FTN(c_poly_create_int)(poly<int> **ths, int *order, int *coefficients) {
        c_poly_create(ths, order, coefficients);
    }
    void FTN(c_poly_evaluate_double)(poly<double> **ths, double *x, double *res) {
        c_poly_evaluate(ths, x, res);
    }
    void FTN(c_poly_evaluate_int)(poly<int> **ths, int *x, int *res) {
        c_poly_evaluate(ths, x, res);
    }
}
}
```

An F90 program using the class:

```
program polyEx
Use polyMod
type(polyPtr) :: poly1
type(polyPtr) :: poly2
real(r8) :: x, res
integer :: order, intx, intres
real(r8) :: coeff1(4) = (/ 4.4, 3.3, 2.2, 1.1 /)
integer :: coeff2(3) = (/ 3, 2, 1 /)
poly1 = polyCreate(3, coeff1)
```

```

poly2 = polyCreate(2, coeff2)

call polyGetOrder(poly1, order)
print *, "order of poly1 is:", order

call polyGetOrder(poly2, order)
print *, "order of poly2 is:", order

do x=-20., 20.,0.5
call polyEvaluate(poly1, x, res)
intx = x
call polyEvaluate(poly2, intx, intres)
print *, "poly1(", x, ") = ", res, "poly2(", intx, ") = ", intres
enddo

end

```

Constructing the interface between the templated C++ class and the F90 module is similar to the approach used in the inheritance example. However, we can no longer use the trick of passing in our `this` pointer to the C++ class as a base class pointer. Now class pointers also have a template parameter that must be resolved, since there is no way in C++ of representing a generic templated type as a pointer; the class has to know the specify type to operate. Hence, we define a set of interface functions that convert the `this` to a typed-pointer and pass the arguments to a member function call, and we create these as parameterized functions. These are not callable yet by F90, because they are only function templates and not actual functions. To finalize the link, we must create a true C function that calls the templated function with the specific arguments that we are going to use (in this example, `double` and `int`).

Some functions within the templated class do not have any real reason to care what the template type is (i.e. they do not use any instances of the type in their algorithm). For instance, `get_order` merely returns an `int` specifying the order of the polynomial. It does not use any `DTYPE` parameter. For these functions, we do not need to explicitly write out every templated type. We can use a place holder. So here we create one interface function:

```
void FTN(c_poly_get_order)(poly<int> **ths, int *order)
```

This function is called for every type, but it still works since the `poly<int>::get_order` function works just as well as the `poly<double>::get_order` function, or any other type, including `void*`. Normally one would use `void*`, but this class requires that the type have basic arithmetic operators and `void*` does not compile.

Also of interest is how the F90 interface mechanism fits nicely with the templating. For example, the function:

```

interface polyCreate
module procedure polyCreateReal
module procedure polyCreateInt
end interface

```

automatically is dispatched in F90 to the appropriate type. Under the hood the C++ interface calls the corresponding template type for either `double` or `int`.

5.2 F90 Implementation

The FORTRAN language does not have templating. Therefore separate algorithms must be coded for every type used, or special-purpose macros written.

5.3 C Implementation

The C language does not have templating. Therefore separate algorithms must be coded for every type used, or special-purpose macros written.

6 Array Sharing

There are a number of issues when trying to share arrays between F90 and C/C++. Besides different storage conventions, F90 has a richer array syntax compared to C's very simple address based system. The two key items below are how to create code that loops through the array without being tied to the storage order, and how to interpret F90 arrays (especially array sections) from C/C++.

6.1 Memory Ordering

Storage schemes in F90 and C are typically reversed. F90 stores multi-dimensional arrays in column-major and C and C++ in row-major order. Being able to access arrays stored in either order is a routine task for numerical libraries. We present an example below to demonstrate some options.

A C++ solution Code for the example is in `template_example/F90`.

The optimal solution to the ordering problem allows the implementation language to use identical code within an algorithm to operate efficiently on data that may be ordered any way. The syntactic “holy grail” is to be able to reference an array (or array object) as `array[i][j]` and have the resulting data item be the same whether the array is passed from F90 (and is thus column-major), or passed from C (and is thus row-major). The semantic richness of C++ allows this to be done by creating an array abstraction class and overloading the correct operators. For random access of an array, this is an immediate solution to the ordering problem. However access through this abstraction adds considerable overhead when compared to the compile time optimized `array[i][j]` that occurs on a true array.

There is still hope for an optimized abstraction, however, because random access is not the usual method of operating on arrays. More common is the iteration through a row, column, or (more generically) a dimension. Take as an example matrix multiplication, matrix inverse, transpose, etc... Each of these algorithms uses an entire dimension, and then moves to the next. The only difference here between column and row major ordering is that in one the stride is *one*, whereas it is the product of all previous dimensions in the other. This is the basis for the iterator abstraction.

We begin by creating an array wrapper/abstraction class, which we can template and re-use for different data types and ranks. When we call a function in our C++ library (matrix multiply is the example here), in our interface we immediately wrap our array:

```
typedef Array<double, 2> double2d;

// Perform mat1*mat2 = matres
void FTN(c_mat_mult)(int *dims, double *mat1, double *mat2, double *matres) {
    double2d arr1(mat1, double2d::F90, dims[0], dims[1]);
    double2d arr2(mat2, double2d::F90, dims[1], dims[2]);
    double2d arrres(matres, double2d::F90, dims[0], dims[2]);
    // ... Use arr1, arr2, arrres for algorithm
}
```

In the C interface we do a similar wrapping, but we pass `double2d::C` as a parameter to the object to signal row-major ordering. Now in our matrix multiply we can concentrate on the algorithm and ignore the complexities of ordering. As a first try we take advantage of C++ operator overloading and use the `operator[]`. The result is identical to our pure C implementation:

```
for (i = 0; i < k; i++) {
    for (j = 0; j < n; j++) {
        res = 0;
        for (l = 0; l < m; l++) {
            res += (double) arr1[j][l] * (double) arr2[l][i];
        }
        arrres[j][i] = res;
    }
}
```

```

}
}

```

Unfortunately the performance of this example is poor, so we introduce a second abstraction mechanism, the dimension iterator. We achieve performance that is not likely to be topped unless the algorithms for column and row ordering are both hand written (a maintenance nightmare):

```

double2d::dim_iterator m1r, m2c, resi;
for (i = 0; i < k; i++) {
    for(resi = matres.dim_begin(double2d::DIM_ITERATE, i), j = 0;
        !resi.done();
        resi++, j++ ) {
        res = 0;
        for (m1r = mat1.dim_begin(j, double2d::DIM_ITERATE),
            m2c = mat2.dim_begin(double2d::DIM_ITERATE, i);
            !m1r.done();
            m1r++, m2c++) {
            res += *m1r * *m2c;
        }
        *resi = res;
    }
}
}

```

The call to `dim_begin` must specify the index of all dimensions except the one to iterate over, which is specified as a wildcard by `DIM_ITERATE`. Calling the `operator++` on the iterator advances by the stride value (the optimal generic solution to the ordering problem). The iterator is overloaded such that dereferencing it accesses the raw array element (note `*resi = res` and `res = *m1r + *m2c`).

6.2 Array Sections

It is common in F90 to pass a subarray to a function call. This is accomplished using the unique F90 array syntax, for example:

```

real*4 :: a(100,100)

call foo(a(2:20:2, 1:10))

```

The above call passes in a section of the original array; every other row from 2 to 20 and the columns 1 to 10. This capability is implemented in F90 with what is commonly called *dope vectors*. This generally means that F90 passes a pointer to a structure describing the array slice. The F90 routine that is passed the array interprets the strides and sizes of each dimension from the table and operates on the array “in place”, saving the overhead of copying the array to a contiguous block, yet allowing the subroutine to still address the array as if it were contiguous.

If the function `foo` (above) is to be implemented in C, there must be a method by which C can discover the various sizes and strides of the array section passed. One method of doing this is the direct interpretation of the *dope vector* in C, i.e.:

```

typedef struct dope_dec{ /* defines dope vector types for CPQ/DEC f90 compiler */
    char rank;           /* Number of dimensions */
    char r[7];           /* Spacer */
    char kind;           /* Either 4 or 8 bytes */
    char k[7];           /* Spacer */
    long addr;           /* Base address of array */
}

```

```

    long d[2];                /* Spacer */
    long ll[MAX_RANK][3]; /* strides (bytes), length(words) and start of each dimension */
}
    DOPEDEC, *DOPEDEC_PTR;

void foo(DOPEDEC_PTR d)
{
...
}

```

When F90 passes an array to another F90 function (or a C function masquerading behind an interface block), a pointer to the structure above is passed. The C function can discover the various sizes and strides of the array from this structure, and the calculations may proceed (you have to be cautious and create an F90 interface block for your C function, otherwise F90 will copy the entire array slice into contiguous memory, causing severe performance degradation).

The weakness of this approach is that each compiler and each platform will have a different structure, and these structures are usually proprietary and only useable through consultation with vendors or experimentation. Keeping track of the various structures on numerous platforms could be difficult and risky.

An alternative solution is to recover the strides and sizes from a set of intermediary F90 functions. This method begins by creating a module which converts F90 arrays to a C++ class handle. The C++ class is an array descriptor, containing the strides and sizes of the array, as well as containing methods to iterate through the array. In this module one function is needed for each array size and type. For instance, there is a function for `real*4` of 1 dimension, another for 2-d, and so on. The same for `integer`, `real*8`, etc... These functions query the language itself for the information that the previous method gleaned from the headers:

```

interface getArrDesc
  module procedure
    arrsec_desc1,
    arrsec_desc2,
    ...
end interface

function arrsec_desc1(a)
type(arraysection) :: arrsec_desc1
real*4 :: a(:)
call arrsec_desc1c(arrsec_desc1,
& size(a), a(1), a(2))
end function

function arrsec_desc2(a)
type(arraysection) :: arrsec_desc2
real*4 :: a(:, :)
call arrsec_desc2c(arrsec_desc2,
& size(a,1), size(a,2),
& a(1,1), a(1,2), a(2,1))
end function

function arrsec_desc3(a)
type(arraysection) :: arrsec_desc3
real*4 :: a(:, :, :)
call arrsec_desc3c(arrsec_desc3,
& size(a,1), size(a,2), size(a,3),
& a(1,1,1), a(1,1,2), a(1,2,1), a(2,1,1))
end function

```

```
end function
```

Listed here are the functions that convert 1-d, 2-d and 3-d real*4 arrays to the C++ descriptors. The function calls a C function which is provided the sizes of each dimension, and then a unit step in each dimension. Since F90 passes scalars by address, each array element is passed as an address and the stride in each dimension can be deduced by simple pointer arithmetic:

```
// C function to dispatch class function
void arrsec_desc3c_(arrsec_desc<float> **arr, int *size1, int *size2, int *size3,
    float *a111, float *a112, float *a121, float *a211)
{
    (*arr) = new arrsec_desc<float>(size1, size2, size3, a111, a112, a121, a211);
}

// C++ class constructor for 3-d case. Class is overloaded for the different possible number
// of dimensions.
arrsec_desc(int *size1, int *size2, int *size3, DTYPE *a111, DTYPE *a112, DTYPE *a121, DTYPE *a211)
{
    dim = 3;

    // Set up the sizes of the dimensions
    dims[0] = *size1;
    dims[1] = *size2;
    dims[2] = *size2;

    // Register the stride lengths
    strides[0] = a112 - a111;
    strides[1] = a121 - a111;
    strides[2] = a211 - a111;

    // Get the base pointer
    base_ptr = a111;
}
}
```

The C++ class contains the data necessary to navigate the array:

```
template<class DTYPE>
class arrsec_desc {
public:
    // 1-D constructor.
    arrsec_desc(int *size, DTYPE *a1, DTYPE *a2);

    // 2-D constructor.
    arrsec_desc(int *size1, int *size2, DTYPE *a11, DTYPE *a12, DTYPE *a21);

    // 3-D constructor.
    arrsec_desc(int *size1, int *size2, int *size3, DTYPE *a111, DTYPE *a112, DTYPE *a121, DTYPE *a211);

    ~arrsec_desc();
};
```

```

float *get_element_ptr(int i);
float *get_element_ptr(int i, int j);
float *get_element_ptr(int i, int j, int k);
int get_num_dims();
int get_dim_size(int i);
private:
    int dim;                // Number of dimensions in array
    int dims[MAX_DIMS];    // Sizes of each dimension
    int strides[MAX_DIMS]; // Stride in each dimension
    DTYPE *base_ptr;       // Address of base array section
};

```

These functions are used in the interface code between F90 and C/C++ as in this example:

Fortran code:

```

real*4 :: a(100,100)

call foo(a(2:20:2, 1:10))

```

Fortran interface code:

```

interface getArrDesc
module procedure,
end interface

subroutine foo(a)
real*4 :: a(:, :)
type(ESMF_ArrayDesc) :: arr

arr = getArrDesc(a)

! Call the C function

call c_foo(ar)

....

```

The implementation C function:

```

void foo(arrsec_desc<float> **arr)
{
/*... Use arr to for some calculation */
}

```

The memory ordering class and the array section class could easily be brought together into a single utility that provides all of the capabilities necessary to handle F90 arrays in C.

7 Pointer Size

The ESMF must support both 32 and 64 bit architectures. Since deep classes are represented on the F90 side by handles, and since we are doing some manual manipulations on pointers, we need to take care to make sure that our classes support both types of pointers.

7.1 C/C++ Implementation

In our C/C++ code we let the compiler decide on the pointer size by making the argument a pointer:

```
void FTN(c_alpine_create)(climber **ths);
```

The C/C++ code dereferences this pointer to pointer and uses the result. On a 32 bit platform the result will be 4 bytes. On a 64 bit platform it will be 8 bytes. The memory that the dereference accesses is actually our F90 type:

```
type climberPtr
sequence
integer(r8) this
end type
```

In this simple example we have made sure that we have at least 8 bytes of memory available so the pointer may be stored. In the more complex examples we have the build process specify the pointer size, and our type looks like:

```
! In config header file
#define POINTER_SIZE 8

type decomp
character, dimension(POINTER_SIZE) :: this
end type
```

8 Function Pointers

Code for the example is in `function_pointers/`.

All of the languages considered in this document contain some type of mechanism for the use of function pointers. C and C++ have an extensive capability, allowing functions to be stored, tabled, and passed as arguments. F90 allows only the last of these capabilities. Once again, by combining these languages, FORTRAN is enhanced so that F90 can have a complete set of operations with function pointers.

8.1 Basic C function Pointers

In C, each function has an address in the code segment. This address may be stored in a pointer (with special syntax), and later invoked from this pointer. This is the mechanism that is commonly used in callbacks and virtual function tables.

An example:

```
void (*event_handler)(Event*) = NULL;

void my_event_handler(Event *event) {
    printf("An event of type:%s occurred\n", event->type);
}
```

```

void do_tasks() {
    Event event;
    /* ..... Do some tasks */
    /* if event occurs */
    if (event_handler != NULL) {
        event_handler(event);
    }
}

main() {
    event_handler = my_event_handler;
    do_tasks();
}

```

An F90 program can use the interface with C to store off F90 functions for later calling. Thus F90 can take advantage of the callback mechanism that is so prevalent in C programs. The callback can be either a C or an F90 function.

The example code demonstrates storing three different sort routines in a callback table. One sort is written in C, another in F90, and another is function within an F90 module. The routine `ctable.c` contains some C routines which provide an API for storing and later calling the sort routines. In the example each sort function takes a two dimensional array and the lengths of each dimension as argument. Each column is considered to be a list of integers to be sorted. The sort routines sorts all columns in place.

```

typedef struct {
    void (*func)(int*, int*,int*);
} VFUNC_TABLE;

VFUNC_TABLE func_table[20];

static int nfuncs = 0;

void FTN(register sortfunc)(void (*func)(int*,int*,int*)) {
    func_table[nfuncs++].func = func;
}

void FTN(call sortfunc)(int *i, int *array, int *nel, int *nell) {
    func_table[*i - 1].func(array, nel, nell);
}

```

The main program is written in F90. It registers the three sort functions, then randomizes a list and calls each sort routine. The functions from `ctable.c` are used to store the function pointers.

```

program fptr
use mod_fptr
integer :: i
integer, target :: int_list(3,100)
integer, pointer :: int_slice(:, :)

! Functions used as pointers must be external (or module)
external csort, f90sort

```

```

        call random_seed()

! Function fills list with random numbers
        call fill_random_list(int_list)

! Demonstrate passing a slice to a sort routine
        int_slice => int_list(1:2, 1:50)

        call print_list(int_list)

! Register the sort functions (i.e. save off pointers)
        call registersortfunc(csort)
        call registersortfunc(f90sort)
        call registersortfunc(f90modsort)

! Demonstrate calling a c sort routine with an array slice
        print *, "Using csort"
        call callsortfunc(1, int_slice,
& size(int_slice,1),size(int_slice,2))
        call print_list(int_list)

        call fill_random_list(int_list)

! Now call an f90 function that has been stored away
        print *, "Using f90sort"
        call callsortfunc(2, int_list, size(int_list,1),
& size(int_list,2))
        call print_list(int_list)

        call fill_random_list(int_list)

! Now call an f90 module function that has been stored away
        print *, "Using f90modsort"
        call callsortfunc(3, int_list,
& size(int_list,1),size(int_list,2))
        call print_list(int_list)

```

This example demonstrates how function pointers may be used in F90 and how F90 functions can be stored as callbacks, using a C library of function pointer routines. The sort example was chosen to demonstrate that such callback functions can be passed arguments, in this case a two dimensional array.

9 Optional Arguments in C++

Code for the example is in `opt_arg.C`. This code demonstrates the ideas below with a different yet similar example.

Optional arguments in F90 are more flexible than their counterparts in C++. Whereas F90 allows the arguments to be specified in any order, C++ demands that the arguments be placed in the same order as the function prototype. However, there is a standard trick to get around this that has been around even prior to C++, and that has been used successfully in C in such libraries as the Xwindows system. For example,

```

/* Example of an Xwindows call to retrieve information about
   a window. */
XtVaGetValues(XtParent(w),
              XmNnumChildren, &numkids,
              XmNchildren, &kids,
              NULL);

```

The call above asks a window to get all of its sibling windows. Windows (as objects) have many attributes (foreground colors, background color, size, stacking behavior, etc...) and, obviously, if only some of these attributes are desired one does not want to have to ask for anything more than what is needed. Yet, at the same time, the programmer does not want to have to provide a different accessor function for every possible combination of attributes a user might want.

The trick is to use C stdargs and a list of name, attribute pairs. The prototype of the above call is,

```

extern void XtVaGetValues(
    Widget          /* widget */,
    ...
);

```

The dots above are the signal that an unknown number of arguments will follow. In the previous call, `XmNnumChildren` and `XmNchildren` are some type of either enumeration or pound defined constants. In the routine, code parses each argument, first getting the `XmNnumChildren` attribute. This tells the code what type of argument to expect next, and this argument can be pulled off the call stack with the `va_arg` call. See `man stdarg` for more details on the use of this library.

The arguments may be sent in any order, since the code parses each pair separately. Also, any number of arguments may be either included or excluded, as they are parsed dynamically. The very last argument in this type of call is typically a `NULL`, since the list needs to know when to quit processing.

10 Performance

Our example suite includes two kinds of examples implemented in C, C++ and F90. The first examples are pure computation and do not include language wrapping or communication. The second kind are more complex, include communication, and are wrapped with C++ and F90 interfaces.

10.1 Single-Language Computation

Code for the example is in `computation_example/mandelbrot`.

Three programs were written in F90, C and in C++ that compute the Mandelbrot set. The code is line to line comparable in the three examples. This test was performed with various compilers across various platforms. The set was calculated on a grid of 1024x768 points. The view into the complex plane is from -2.0 to 2.0 on the real and imaginary axis. The test created the set 20 times for each run, and each run was performed twice to make sure the results were comparable from run to run. The results were averaged. All examples were compiled with `-O3`. The averages per run are:

Program/Compiler	C (seconds)	C++	F90
Linux (pgCC, pgf90)	26.5	26.5	26.6
SGI (vendor compiler)	49.5	49.6	40.9
IBM rs6000 (vendor)	34.2	21.0	20.9
Sun (vendor compiler)	46.0	32.1	51.3

The conclusion is that performance is generally very close. On most of the platforms F90 was slightly faster, but on one (the Sun) it was slower. A huge difference was seen when putting the GNU compilers head to head with the

Portland Group compilers. The GNU tools were almost three times as slow. A huge difference between C/C++ and F90 is seen when compiling in debug mode (which should not be a problem).

10.2 Large-Scale Example

Code for the example is in `large_scale_example/mandelbrot`.

An example drawn from the NASA DAO PILGRIM library demonstrates the performance of various implementations of a procedure involving regular data transfers. The exercise takes a 2048x5 array of gridpoints and creates a 2d square decomposition of the points over 9 (3x3) processors. The gridpoints are then paraded around the decomposition in a circular manner 100 times using a transpose, until they finally reach their original position. A comparison of buffers at the end of the programs verifies that the gridpoints made it correctly back to their original position. This example utilizes a larger code base than the computation example and involves MPI communication.

The following table shows times in seconds for C++ wrapped in F90 and for the original F90 PILGRIM library implementation. The C++ implementation is considerably faster.

Program/Compiler	F90	C++
SGI (vendor compiler)	41.0	5.5
IBM rs6000 (vendor)	7.2	3.1
Sun (vendor compiler)	35.0	4.2

The codes are not line-to-line comparable. There were a number of optimizations that were easily done in the C++ code that contributed to its speed. For instance, a permutation is done by simply creating an indexing array, whereas PILGRIM copies the entire array.

11 Linking C++ and F90

Linking F90 with C++ can be challenging. Linking C++ with anything can be challenging. However, it is possible, though there is no generic method that works on all platforms. Many of the platforms have problems with the entry point, since traditionally the F90 entry is `MAIN_`, where the C++ entry is `main`. However, using link flags these difficulties can be overcome. On some platforms there is no problem. Template instantiation can be a problem, and the difficulties are directly proportional to how smart the C++ compiler tries to be. Most of these overzealous compilers can be told to relax via compiler flags. In general F90 is more well behaved, and this helps. On all the compilers we used, the other language's standard libraries were not found. This problem can be solved by running the opposite compiler in verbose mode (`-v` on many platforms). It then shows its `ld` line, and the libraries may be ascertained.

11.1 SunOS

The sun MPI compilers are `/optSUNWhpc/bin/[mpf90, mpCC, mpcc]`. We have found it easiest to link on this platform using `mpCC`. This C++ compiler will not link F90 code immediately. However, when passed the correct libraries it becomes satisfied. They were:

```
-lfui -lfai -lfai2 -lfsumai -lfprodai -lfminlai -lfmaxlai -lfminvai -lfmaxvai
-lfsu -lsunmath -lm -lc
```

It was quite important on this platform to turn off the C++ compiler's "smart" templates. This feature tries to gather instances of templates at link time, and prevents linking from occurring. To do this, the compiler requires the flag `'-instantiate=static'`, which tells it to act like the original C++ compilers and just put the instances of a template used into each object file.

11.2 AIX

The IBM compilers (surprisingly) adapt to linking C++ and F90 quite easily. We used the C++ compiler and merely had to send it the F90 libraries. These were:

```
-L/usr/lpp/ppe.poe/lib/threads -L/usr/lpp/ppe.poe/lib  
-L/usr/lpp/ppe.poe/lib/ip,-L/lib/threads -lmpi_r -lvtd_r -lxlf90 -lxlopt -lxlf  
-lxlomp_ser -lpthreads -lm_r -lm -lc_r -lc /usr/lpp/ppe.poe/lib/libc.a
```

11.3 IRIX

Irix C++ has the same officiousness as does the Sun compiler. Its attempt at being clever with C++ templates can be stopped easily by sending the `-no_prelink -ptused` flags, which tell the compiler not to pre-link, and instantiate all templates that are referenced.

11.4 Linux

The linux configuration of `g++` and `pgf90` worked together quite well. The linker used was `pgf90`, and the only C++ library that had to be added to the link line was `-lstdc++`.

12 Using Shared Libraries

By compiling the code and linking it into a shared library, the addition of the library flags mentioned above can be hidden from the user so that they only have to add `-lesmf -Lpath_to_esmf` to their link line. This is because the shared object is “prelinked”, whereas an archive library is only a collection of object files. The library users have to point their F90 compiler to the F90 module directory.

13 Tools

Most of the interface code in the above examples was hand coded to fit the example. However, language interoperability is an increasingly common concern and a number of tools are being currently developed that can make the above strategies automated and more standard.

13.1 Chasm

Chasm is a relatively new software project at Los Alamos National Laboratory that bridges the interoperability divide between Fortran and C++. Chasm tools parse Fortran (or C++) source files using commercial quality, standards-compliant compiler front-ends (these compiler tools are provided free of charge by the compiler vendors) and automatically generate bridging code for language interoperability. Chasm is designed to be extensible so that the actual contents of the bridging code can be chosen by the users of Chasm. This allows the style of language interoperability to be chosen and fixed on a project-wide basis. Finally, Chasm generates code to be incorporated into a project, so there need not be a long-term dependency on Chasm.

Chasm solves many of the routine problems encountered in language interoperability. The first problem is one of simple maintenance. As interfaces evolve in the implementation language, the wrappers for the alternate language can become out of synch with the implementation. The second problem is one of standardization. There are multiple ways to approach language interoperability and it is likely that more than one style would likely be used unless an interoperability coding standard is strictly enforced. Multiple interoperability coding styles inherently lead to confusion and ultimately to errors. Finally, there is the simple drudgery of manually creating wrappers for many separate software interfaces. In summary, the manual creation and maintenance of bridging code between C++ and Fortran is expensive and error prone.

Chasm reduces the expense of creating and maintaining common interfaces in Fortran and C++ and allows an interoperability coding policy to be set and enforced. The Chasm/ESMF interaction over the past year has been positive for both groups; as an early “friendly user” of Chasm the ESMF group was able to install and test Chasm on NCAR machines and offer feedback on its design. This interaction led to an important addition to the Chasm design, namely the ability to customize its interoperability model. The ESMF group anticipates using Chasm’s interpretation

of dope vectors for well-known compilers and the more generic (and slower) array sharing strategy described in Section 6.2 for unknown compilers.

We will continue to track Chasm progress and hope to start using Chasm to automate language interface generation in spring 2003.

13.2 Babel

Babel [?] is a tool for achieving interlanguage interoperability between a wide variety of programming languages. Babel's approach is based on interface definition language (IDL) techniques, through which a user describes the calling interfaces of a software package, but not their implementation. Babel has its own IDL called the Scientific Interface Definition Language (SIDL). SIDL supports intrinsic types, complex numbers, and multidimensional arrays, and provides an object-oriented inheritance model similar to Java. The Babel package consists of a SIDL parser, a code generator, a small run-time support library, and the Alexandria Web interface and component repository. Users need only use the first three of these elements to construct interlanguage interfaces. Languages currently supported by Babel include C++, C, Java, Fortran77, and Python.

The SIDL parser reads a user-generated SIDL description of the package to be exported to other languages, and creates an XML representation of the SIDL interfaces. Babel's code generator uses the XML interface representation to generate the necessary glue code, which comprises: *implementation prototypes* (known colloquially as "IMPL files"), an *internal object representation* (IOR), stubs, and skeletons. The Babel IOR is generated in C, and is essentially a table of function pointers for the methods for a given class. IMPL files produced by Babel contain function prototypes and splicer blocks defining where the implementation (calls to the user's code) are filled in by the developer. Babel also aids the developer by providing automatic generation of Makefiles, and uses the aforementioned splicer blocks to preserve previous work in the event of extension or modification of the SIDL file. Stubs and skeletons translate between the calling conventions of supported languages and the Babel IOR. Babel's run-time arguments allow the user to create only certain portions of the glue code at a time.

A user wishing to export a package for use by other languages begins by describing its interfaces in a SIDL file. The user then runs Babel, which parses the SIDL file and creates an XML representation of the SIDL interfaces, and a template IMPL file in the package's implementation language. The user then fills in the IMPL file with the necessary code to use the package, along with the necessary Makefiles. Interfaces for target languages (which the user can choose via run-time options) and their associated Makefiles are then generated by Babel.

Babel is under development, and only partial support for Fortran90 is currently available. This fact has forced ESMF and others in the climate community to adopt a "wait and see" attitude towards Babel. The need for Fortran90 support for the climate community motivated a re-ordering of priorities for the Babel development team, with Fortran90 support a top priority. They are working to leverage infrastructure from CHASM, and full support for Fortran90 is expected in mid-to-late 2003. Once full Fortran90 support is provided by Babel, we will perform some simple experiments to evaluate its applicability and utility to the ESMF project.

14 Conclusions

Our examples and experiments indicate that it is feasible to develop a high-performance framework with both C++ and F90 interfaces. Interestingly, our examples also show that when C++ code underlies F90, some of the advantages of the C++ language can migrate across the language interface and enhance the capabilities of F90.

We will adopt a mixed-language implementation for the ESMF. Utility classes will be implemented in C++, and field and grid classes in F90. Because inheritance and function pointers are integral to the design of the superstructure, we anticipate using C++ in its implementation. In order to use the capabilities of the framework, scientists will not need to understand C++ and will (if they so desire) be able to interact strictly with a natural F90 interface.

We have decided to write portions of the framework in C++ because it offers better support than F90 for useful language features such as templating, inheritance, function pointers, and polymorphism.

We have decided to write field and grid classes in F90 in part because these are the structures scientists are most interested in examining in detail, modifying, and extending, and F90 is the language most Earth scientists are familiar

with. We also have a large body of existing F90 code, SCRIP (see [?]), on which we will be building the ESMF regridting services. Although field and grid classes will be implemented in F90, we plan to develop prototypes of these classes in C++ during the period of NASA funding in order to further explore issues of performance and flexibility.

15 Glossary

shallow class A library class that has no memory allocated by the library.

deep class A library class in which the library allocates the class memory.

opaque object A class whose data and attributes must be accessed through a function call.