Compiling and Linking and Interfacing Multiple Programming Languages

Dr. Axel Kohlmeyer

Research Professor, Dept. of Mathematics
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

Pre-process / Compile / Link

- Creating an executable includes multiple steps
- The “compiler” (gcc) is a wrapper for <u>several</u> commands that are executed in succession
- The “compiler flags” similarly fall into categories and are handed down to the respective tools
- The “wrapper” selects the compiler language from source file name, but links “its” runtime
- We will look into a C example first, since this is the language the OS is (mostly) written in

A simple C Example

- Consider the minimal C program 'hello.c':

```
#include <stdio.h>
int main(int argc, char **argv)
{
        printf(“hello world\n”);
        return 0;
}
```

- i.e.: what happens, if we do:

  > **`gcc -o hello hello.c`**

  (try: **`gcc -v -o hello hello.c`**)

Step 1: Pre-processing

- Pre-processing is <u>mandatory</u> in C (and C++)
- Pre-processing will handle '#' directives
  - File inclusion with support for nested inclusion
  - Conditional compilation and Macro expansion
- In this case: **`/usr/include/stdio.h`**
  - and all files are included by it - are inserted and the contained macros expanded
- Use -E flag to stop after pre-processing:
  > **`cc -E -o hello.pp.c hello.c`**

Step 2: Compilation

- Compiler converts a high-level language into the specific instruction set of the target CPU
- Individual steps:
  - Parse text (lexical + syntactical analysis)
  - Do language specific transformations
  - Translate to internal representation units (IRs)
  - Optimization (reorder, merge, eliminate)
  - Replace IRs with pieces of assembler language
- Try:> **`gcc -S hello.c`** (produces **`hello.s`**)

Compilation cont'd

```
        .file   "hello.c"
        .section    .rodata
.LC0:
        .string "hello, world!"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.1 20100924 (Red Hat 4.5.1-4)"
        .section        .note.GNU-stack,"",@progbits
```

gcc replaced `printf` with `puts`

try: gcc -fno-builtin -S hello.c

```
#include <stdio.h>
int main(int argc,
         char **argv)
{
 printf(“hello world\n”);
 return 0;
}
```

Step 3: Assembler / Step 4: Linker

- Assembler (as) translates assembly to binary
  - Creates so-called object files (in ELF format)

  Try: > `gcc -c hello.c`
  Try: > `nm hello.o`
  ```
  00000000 T main
           U puts
  ```
- Linker (ld) puts binary together with startup code and required libraries
- Final step, result is executable.
  Try: > `gcc -o hello hello.o`

Adding Libraries

- Example 2: exp.c

```
#include <math.h>
#include <stdio.h>
int main(int argc, char **argv)
{    double a=2.0;
     printf("exp(2.0)=%f\n", exp(a));
     return 0;
}
```

- `> gcc -o exp exp.c`

  Fails with “undefined reference to 'exp'”. Add: -lm

- `> gcc -O3 -o exp exp.c`

  Works due to inlining at high optimization level.

Symbols in Object Files & Visibility

- Compiled object files have multiple sections and a symbol table describing their entries:
  - “Text”: this is executable code
  - “Data”: pre-allocated variables storage
  - “Constants”: read-only data
  - “Undefined”: symbols that are used but not defined
  - “Debug”: debugger information (e.g. line numbers)
- Entries in the object files can be inspected with either the “nm” tool or the “readelf” command

Example File: visbility.c

```
static const int val1 = -5;
const int val2 = 10;
static int val3 = -20;
int val4 = -15;
extern int errno;

static int add_abs(const int v1, const int v2) {
    return abs(v1)+abs(v2);
}

int main(int argc, char **argv) {
     int val5 = 20;
     printf("%d / %d / %d\n",
            add_abs(val1,val2),
            add_abs(val3,val4),
            add_abs(val1,val5));
     return 0;
}
```

```
nm visibility.o:
00000000 t add_abs
         U errno
00000024 T main
         U printf
00000000 r val1
00000004 R val2
00000000 d val3
00000004 D val4
```

What Happens During Linking?

- Historically, the linker combines a “startup object” (crt1.o) with all compiled or listed object files, the C library (libc) and a “finish object” (crtn.o) into an executable (a.out)
- With current compilers it is more complicated
- The linker then “builds” the executable by matching undefined references with available entries in the symbol tables of the objects
- crt1.o has an undefined reference to “main” thus C programs start at the main() function

Static Libraries

- Static libraries built with the "ar" command are collections of objects with a global symbol table
- When linking to a static library, object code is copied into the resulting executable and all direct addresses recomputed (e.g. for "jumps")
- Symbols are resolved "from left to right", so circular dependencies require to list libraries multiple times or use a special linker flag
- When linking only the name of the symbol is checked, not whether its argument list matches

Shared Libraries

- Shared libraries are more like executables that are missing the main() function
- When linking to a shared library, a marker is added to load the library by its “generic” name (soname) and the list of undefined symbols
- When resolving a symbol (function) from shared library all addresses have to be recomputed (relocated) on the fly.
- The shared linker program is executed first and then loads the executable and its dependencies

Differences When Linking

- Static libraries are fully resolved “left to right”; circular dependencies are only resolved between explicit objects or inside a library
-> need to specify libraries multiple times
or use: **-Wl,--start-group (...) -Wl,--end-group**
- Shared libraries symbols are **not** fully resolved at link time, only checked for symbols required by the object files. **Full check** only at runtime.
- Shared libraries may depend on other shared libraries whose symbols will be globally visible

Dynamic Linker Properties

- Linux defaults to dynamic libraries:
  ```
  > ldd hello
  linux-gate.so.1 =>  (0x0049d000)
  libc.so.6 => /lib/libc.so.6
  (0x005a0000)
  /lib/ld-linux.so.2 (0x0057b000)
  ```
- **`/etc/ld.so.conf, LD_LIBRARY_PATH`** define where to search for shared libraries
- **`gcc -Wl,-rpath,/some/dir`** will encode **`/some/dir`** into the binary for searching

Using LD_PRELOAD

- Using the LD_PRELOAD environment variable, symbols from a shared object can be preloaded into the global object table and will <u>override</u> those in later resolved shared libraries
  => replace specific functions in a shared library
- Example: override log() with a faster version:
  ```
  #include “amdlibm.h”
  double log(double x) { return amd_log(x); }
  ```
  gcc -shared -o fasterlog.so faster.c -lamdlibm
- LD_PRELOAD=./fasterlog.so ./myprog-with

Before LD_PRELOAD

```
PerfTop:    8016 irqs/sec  kernel: 9.9%  exact:  0.0% [1000Hz cycles],  (all, 8 CPUs)
---------------------------------------------------------------------------------------

             samples  pcnt function                  DSO
             _______ _____ _________________________ _____________________________________

            53462.00 52.2% __ieee754_log             /lib64/libm-2.12.so
            10490.00 10.3% R_binary                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             8704.00  8.5% clear_page_c              [kernel.kallsyms]
             5737.00  5.6% __ieee754_exp             /lib64/libm-2.12.so
             4645.00  4.5% math1                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             3070.00  3.0% __log                     /lib64/libm-2.12.so
             3020.00  3.0% __isnan                   /lib64/libc-2.12.so
             2094.00  2.0% R_gc_internal             /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1643.00  1.6% do_summary                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1251.00  1.2% __isnan@plt               /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1210.00  1.2% real_relop                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1161.00  1.1% __GI___exp                /lib64/libm-2.12.so
              754.00  0.7% __isnan                   /lib64/libm-2.12.so
              739.00  0.7% R_log                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              553.00  0.5% __kernel_standard         /lib64/libm-2.12.so
              550.00  0.5% do_abs                    /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              462.00  0.5% __mul                     /lib64/libm-2.12.so
              439.00  0.4% coerceToReal              /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              413.00  0.4% finite                    /lib64/libm-2.12.so
              358.00  0.3% log@plt                   /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              182.00  0.2% get_page_from_freelist    [kernel.kallsyms]
              120.00  0.1% __alloc_pages_nodemask    [kernel.kallsyms]
```

After LD_PRELOAD

```
  PerfTop:    8020 irqs/sec  kernel:17.2%  exact:  0.0% [1000Hz cycles],  (all, 8 CPUs)
------------------------------------------------------------------------------------------

             samples  pcnt function                  DSO
             _______ _____ _________________________ _____________________________________

            24702.00 19.5% __amd_bas64_log           /opt/libs/fastermath-0.1/libamdlibm.so
            22270.00 17.6% R_binary                  /opt/binf/R-2.13.0/lib64/R/bin/exec/R
            18463.00 14.6% clear_page_c              [kernel.kallsyms]
            10480.00  8.3% __ieee754_exp             /lib64/libm-2.12.so
             9834.00  7.8% math1                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             9155.00  7.2% log                       /opt/libs/fastermath-0.1/fasterlog.so
             6269.00  5.0% __isnan                   /lib64/libc-2.12.so
             4214.00  3.3% R_gc_internal             /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             3074.00  2.4% do_summary                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2285.00  1.8% real_relop                /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2257.00  1.8% __isnan@plt               /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             2076.00  1.6% __GI___exp                /lib64/libm-2.12.so
             1346.00  1.1% R_log                     /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1213.00  1.0% do_abs                    /opt/binf/R-2.13.0/lib64/R/bin/exec/R
             1075.00  0.8% __kernel_standard         /lib64/libm-2.12.so
              894.00  0.7% coerceToReal              /opt/binf/R-2.13.0/lib64/R/bin/exec/R
              780.00  0.6% __mul                     /lib64/libm-2.12.so
              756.00  0.6% finite                    /lib64/libm-2.12.so
              729.00  0.6% amd_log@plt               /opt/libs/fastermath-0.1/fasterlog.so
              706.00  0.6% amd_log                   /opt/libs/fastermath-0.1/libamdlibm.so
              674.00  0.5% log@plt                   /opt/binf/R-2.13.0/lib64/R/bin/exec/R
```

Difference Between C and Fortran

- Basic compilation principles are the same
  => preprocess, compile, assemble, link
- In Fortran, symbols are case insensitive
  => most compilers translate them to lower case
- In Fortran symbol names may be modified to make them different from C symbols
  (e.g. append one or more underscores)
- Fortran entry point is not “main” (no arguments)
  PROGRAM => MAIN__ (in gfortran)
- C-like main() provided as startup (to store args)

Pre-processing in C and Fortran

- Pre-processing is <u>mandatory</u> in C/C++
- Pre-processing is <u>optional</u> in Fortran
- Fortran pre-processing enabled implicitly via file name: name.F, name.F90, name.FOR
- Legacy Fortran packages often use /lib/cpp:
  /lib/cpp -C -P -traditional -o name.f name.F
  - -C : keep comments (may be legal Fortran code)
  - -P : no '#line' markers (not legal Fortran syntax)
  - -traditional : don't collapse whitespace (incompatible with fixed format sources)

Fortran Symbols Example

```
 SUBROUTINE GREET
   PRINT*, 'HELLO, WORLD!'
END SUBROUTINE GREET

program hello
  call greet
end program
```

```
0000006d t MAIN__
         U _gfortran_set_args
         U _gfortran_set_options
         U _gfortran_st_write
         U _gfortran_st_write_done
         U _gfortran_transfer_character
00000000 T greet_
0000007a T main
```

- “program” becomes symbol “MAIN__” (compiler dependent)
- “subroutine” name becomes lower case with '_' appended
- several “undefineds” with '_gfortran' prefix
  => calls into the Fortran runtime library, libgfortran
- cannot link object with “gcc” alone, need to add -lgfortran
  => cannot mix and match Fortran objects from different compilers

Fortran 90+ Modules

- When subroutines or variables are defined inside a module, they have to be hidden

```
module func
  integer :: val5, val6
contains
  integer function add_abs(v1,v2)
    integer, intent(in) :: v1, v2
    add_abs = iabs(v1)+iabs(v2)
  end function add_abs
end module func
```

- gfortran creates the following symbols:

```
00000000 T __func_MOD_add_abs
00000000 B __func_MOD_val5
00000004 B __func_MOD_val6
```

The Next Level: C++

- In C++ functions with different number or type of arguments can be defined (overloading) => encode prototype into symbol name:

  Example : symbol for `int add_abs(int,int)` becomes: `_ZL7add_absii`

- Note: the return type is <u>not</u> encoded
- C++ symbols are no longer compatible with C => add 'extern “C”' qualifier for C style symbols
- C++ symbol encoding is <u>compiler specific</u>

C++ Namespaces and Classes vs. Fortran 90 Modules

- Fortran 90 modules share functionality with classes and namespaces in C++
- C++ namespaces are encoded in symbols
  Example: `int func::add_abs(int,int)`
  becomes: `_ZN4funcL7add_absEii`
- C++ classes are encoded the same way
- Figuring out which symbol to encode into the object as undefined is the job of the compiler
- When using the gdb debugger use '::' syntax

Why We Need Header or Module Files

- The linker is "blind" for any language specific properties of a symbol => checking of the validity of the interface of a function is only possible during compilation
- A header or module file contains the prototype of the function (not the implementation) and the compiler can compare it to its use
- Important: header/module has to match library => Problem with FFTW-2.x: cannot tell if library was compiled for single or double precision

Calling C from Fortran 77

- Need to make C function look like Fortran 77
  - Append underscore (except on AIX, HP-UX)
  - Call by reference conventions
  - Best only used for “subroutine” constructs (cf. MPI) as passing return value of functions varies a lot:
    ```
    void add_abs_(int *v1,int *v2,int *res){
    *res = abs(*v1)+abs(*v2);}
    ```
- Arrays are always passed as “flat” 1d arrays by providing a pointer to the first array element
- Strings are tricky (no terminal 0, length added)

Calling C from Fortran 77 Example

```
void sum_abs_(int *in, int *num, int *out) {
 int i,sum;
 sum = 0;
 for (i=0; i < *num; ++i) { sum += abs(in[i]);}
   *out = sum;
   return;
}

/* fortran code:
   integer, parameter :: n=200
   integer :: s, data(n)

   call SUM_ABS(data, n, s)
   print*, s
*/
```

Calling Fortran 77 from C

- Inverse from previous, i.e. need to add underscore and use lower case (usually)
- Difficult for anything but Fortran 77 style calls since Fortran 90+ features need extra info
  - Shaped arrays, optional parameters, modules
- Arrays need to be “flat”,
  C-style multi-dimensional arrays are lists of pointers to individual pieces of storage, which may not be consecutive
  => use 1d and compute position

Calling Fortran 77 From C Example

```
subroutine sum_abs(in, num, out)
   integer, intent(in)  :: num, in(num)
   integer, intent(out) :: out
   Integer              :: i, sum
   sum = 0
   do i=1,num
     sum = sum + ABS(in(i))
   end do
   out = sum
end subroutine sum_abs
!! c code:
!   const int n=200;
!   int data[n], s;
!   sum_abs_(data, &n, &s);
!   printf("%d\n", s);
```

Modern Fortran vs C Interoperability

- Fortran 2003 introduces a standardized way to tell Fortran how C functions look like and how to make Fortran functions have a C-style ABI
- Module “iso_c_binding” provides kind definition: e.g. C_INT, C_FLOAT, C_SIGNED_CHAR
- Subroutines can be declared with “BIND(C)”
- Arguments can be given the property “VALUE” to indicate C-style call-by-value conventions
- String passing tricky, needs explicit 0-terminus

Calling C from Fortran 03 Example

```
int sum_abs(int *in, int num) {
  int i,sum;
  for (i=0,sum=0;i<num;++i) {sum += abs(in[i]);}
  return sum;
}
/* fortran code:
  use iso_c_binding, only: c_int
  interface
    integer(c_int) function sum_abs(in, num) bind(C)
      use iso_c_binding, only: c_int
      integer(c_int), intent(in) :: in(*)
      integer(c_int), value :: num
    end function sum_abs
  end interface
  integer(c_int), parameter :: n=200
  integer(c_int) :: data(n)
  print*, SUM_ABS(data,n)  */
```

Calling Fortran 03 From C Example

```
subroutine sum_abs(in, num, out) bind(c)
   use iso_c_binding, only : c_int
   integer(c_int), intent(in)  :: num,in(num)
   integer(c_int), intent(out) :: out
   integer(c_int),             :: i, sum
   sum = 0
   do i=1,num
     sum = sum + ABS(in(i))
   end do
   out = sum
end subroutine sum_abs

!! c code:
!   const int n=200;
!   int data[n], s;
!   sum_abs(data, &n, &s);
!   printf("%d\n", s);
```

Linking Multi-Language Binaries

- Inter-language calls via mutual C interface only due to name “mangling” of C++ / Fortran 90+ => extern “C”, ISO_C_BINDING, C wrappers
- Fortran “main” requires Fortran compiler for link
- Global static C++ objects require C++ for link => avoid static objects (good idea in general)
- Either language requires its runtime for link
  => GNU: -lstdc++ and -lgfortran
  => Intel: “its complicated” (use -# to find out)
  more may be needed (-lgomp, -lpthread, -lm)

Dynamic Linking via dlopen()

- POSIX compliant C libraries allow loading of shared objects are runtime via dlopen()/dlsym()
  - Calls to dlopen() open a handle to shared object; lookup of this file is subject to same rules as dynamic library searches
  - Calls to dlsym() look up symbol by its name in shared object pointed to by handle; returns pointer; for functions need to cast/assign to function pointer
  - Calls to dlclose() unload shared object (if last user) and revoke assignments to code made by dlsym()

Example: static program test-0.c

```
#include <stdio.h>

void hello()
    puts(“Hello, World”);
}

int main(int argc, char **argv)
{
    void (*hi)();  /* function pointer variable */

    hi = &hello;   /* initialize function pointer */

    (*hi)();       /* this is the same as: hello(); */
    return 0;
}

/* compile with: gcc -o test-0 -Wall -O test-0.c */
```

Example: main program test-1.c

```
#include <dlfcn.h>

int main(int argc, char **argv)
{
    void *handle;     /* handle for dynamic object */
    void (*hi)();     /* function pointer for symbol */

    handle = dlopen("./hello.so", RTLD_LAZY);
    if (handle) {
       hi = (void (*)()) dlsym(handle,"hello");
       (*hi)();
       dlclose(handle);
    }
    return 0;
}
/* compile with: gcc -o test-1 -Wall -O test-1.c -ldl
 add -rdynamic if shared object needs symbols in main */
```

Example: shared object hello.c

```
#include<stdio.h>

void hello(void)
{
    puts("Hello, World!");
}
/*
 compile: gcc -shared -o hello.so -fPIC -Wall -O hello.c
 */
```

- With this setup, `hello.c` can be changed and `hello.so` recompiled without having to recompile and re-link `test-1`.
- Thus access to `test-1.c` is not needed.

Extending Python with `ctypes`

- The `ctypes` module in python provides an interface to dlopen()/dlsym() and thus allows to call compiled C code from python.
- Support for dll files on Windows is also included
- Since symbols in compiled objects have no information about calling sequence and return values, this has to be set on the python side
- Incorrect use can lead to segmentation faults or corrupted data; often prototypes are needed

Example: calling hello.c from python

```
#!/usr/bin/env python

from ctypes import *

# import shared object on POSIX compatible OS
dso = CDLL(“./hello.so”)

# call symbol in shared object as function w/o args
dso.hello()
```

- This python script does pretty much the same thing as the `test-1` compiled program
- Since there are no arguments and no return values, no code needs to know about the other

Arguments & Return Value

- By default ctypes will assume arguments and return values are standard size integer

```
#include<stdio.h>

int sum_of_int(int a, int b) {
    int c = a + b;
    printf("sum of %d and %d is %d\n",a,b,c);
    return c;
}
--------------------------------------------------
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
isum = dso.sum_of_int(1,2)
print "Integer sum is: ", isum
```

Prototypes with ctypes

- If argument and/or return value are of different type, ctypes needs to be informed about it; works similar to prototypes in C

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
dso.sum_of_int.argtypes = [ c_int, c_int ]
dso.sum_of_int.restype = c_int
isum = dso.sum_of_int(1,2)
print ("Integer sum w/ prototypes is: ", isum)

dso.sum_of_double.argtypes = [ c_double, c_double ]
dso.sum_of_double.restype = c_double
dsum = dso.sum_of_double(0.5,2.5)
print ("Double sum w/ prototypes is: ", dsum)
```

Passing Strings

- Strings in python are read-only, thus when a C-function will modify a string we have to use `create_string_buffer()`

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./hello.so”)

# hello() in hello.so takes a “char *” argument
dso.hello.argtypes = [ c_char_p ]
dso.hello(b”World”)

# create buffer for mutable string data
buf = create_string_buffer(b”World”)
dso.hello(buf)
```

Passing Arrays

- When passing allocatable objects like arrays, it is usually best to do the allocating in python. ctypes offers constructors for all basic types

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./sum.so”)
num = 10
dlist = (c_double * num)()  # (primitive * length)()
for i in range(num):
   dlist[i] = 0.333*(i*0.5)
# note the use of POINTER()
dso.sum_of_doubles.argtypes=[POINTER(c_double),c_int]
dso.sum_of_doubles.restype = c_double
dsum = dso.sum_of_doubles(dlist,num)
print ("Double sum is: ", dsum)
```

Passing Structs

- Even complex storage elements like `struct` can be managed by ctypes. Derive a class from Structure that mimics the corresponding C-type

```
#!/usr/bin/env python
from ctypes import *
dso = CDLL(“./data.so”)

class parm(Structure):
  _fields_ = [ ("type", c_int), ("label", c_char_p),
               ("epsilon",c_double),("sigma",c_double) ]
# use constructor to initialize struct
p = parm(type=1,label=b"LJ-12-6",epsilon=0.1,sigma=3.4)
# p is passed by value, to pass by reference use byref(p)
dso.pass_by_value(p)
dso.pass_by_reference(byref(p))
```

Passing Structs (2)

- Below is the corresponding C code:

```
#include<stdio.h>

struct parm { int  type; char *label;
              double epsilon, sigma;
};

void pass_by_value(struct parm p) {
    printf("type=%d  label=%s epsilon=%g  sigma=%g\n",
           p.type, p.label, p.epsilon, p.sigma);
}

void pass_by_reference(struct parm *p) {
    printf("type=%d  label=%s epsilon=%g  sigma=%g\n",
           p->type, p->label, p->epsilon, p->sigma);
}
```

Interfacing Fortran with f2py

- Interfacing Fortran with python is both easier and more complicated than interfacing C
  - The Fortran ABI can be much more complex and is more compiler specific than the C ABI
  - The numpy project has a tool “f2py” that automates the process and hides the complications
- If you have a fortran file with some functions or subroutine do: f2py -c code.f90 -m module
  - Creates python loadable module “module”
  - Flag '-c' calls compiler; flag '-m' sets module name

Interfacing Fortran with f2py (2)

- Then in python do: from module import *
  and call the Fortran functions in python
- The f2py tool will parse the fortran code and generate the necessary C-code for a module
- The f2py generated code will automatically insert code to convert data as needed;
  e.g. lists are converted to arrays
- The f2py tool works best with well formed Fortran code; otherwise data maps can help

Fortran examples for f2py

- Example code that converts cleanly with f2py:

```
subroutine hello
    print*, "Hello, World!"
end subroutine hello
function sum_of_int(a,b) result(c)
    integer, intent(in) :: a, b
    integer :: c
    c = a + b
    print*,"sum of ", a, " and ", b, " is ", c
end function sum_of_int
function sum_of_double(a,b) result(c)
    double precision, intent(in) :: a, b
    double precision :: c
    c = a + b
    print*,"sum of ", a, " and ", b, " is ", c
end function sum_of_double
```

Passing arrays with f2py

- Arrays are traditional style arrays with f2py:

```
function sum_of_doubles(a,n) result(s)
    double precision, intent(in) :: a(*)
    integer, intent(in) :: n
    double precision :: s
    integer :: i
    s = 0
    do i=1,n
        s = s + a(i)
    end do
end function sum_of_doubles
------------------------------------------------------
num = 10
dlist = [sqrt(float(i)) for i in range(1,num)]
dsum = sum_of_doubles(dlist,num)
```

Passing strings with f2py

- Strings are handled in a very similar fashion to traditional style arrays with f2py:

```
subroutine hello(name)
    character(len=*), intent(in) :: name
    print* , "Hello, ", name, "!"
end subroutine hello
----------------------------------------
#!/usr/bin/env python
from  hello import *

print ("Calling DSO")
hello('World')
print ("Done")
```

Compiling and Linking and Interfacing Multiple Programming Languages

Dr. Axel Kohlmeyer

Research Professor, Dept. of Mathematics
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu