# Mixing C, C++, and Fortran


## Axel Kohlmeyer

Associate Dean for Scientific Computing, CST
Associate Director, Institute for Comput. Molecular Science
Assistant Vice President for High-Performance Computing

**a.kohlmeyer@temple.edu**


ICMS
Institute for Computational Molecular Science

TEMPLE UNIVERSITY®

# 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
```

ICMS
Institute for Computational Molecular Science

# 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)

# 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 not encoded
- C++ symbols are no longer compatible with C => add 'extern “C”' qualifier to have C++ export C style symbols (=> no overloading possible)
- C++ symbol encoding is compiler specific

# 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 still tricky, add 0-terminus for C

# 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)

# Mixing C, C++, and Fortran


## Axel Kohlmeyer

Associate Dean for Scientific Computing, CST
Associate Director, Institute for Comput. Molecular Science
Assistant Vice President for High-Performance Computing

**a.kohlmeyer@temple.edu**


ICMS
Institute for Computational Molecular Science

TEMPLE UNIVERSITY®