Introduction to Modern Fortran

Interoperability with C

Nick Maclaren

nmm1@cam.ac.uk

March 2014
Mixed-language programming is ancient technology
Traditionally done by non-portable hacking and worse

Fortran 2003 has defined a proper interface to C
Extended in TS 29113 – mentioned later
But the old rule number one still holds:

• KISS – Keep It Simple and Stupid

Be ‘clever’ and your program will go wrong
Probably not while debugging, but in actual use
Why Interoperate with C? (1)

Often to get access to system interfaces
Or to extend the intrinsic functions
• Functions are typically very simple in both cases

E.g. get high-precision (microsecond) timestamp
Get environment variable, or invoke command
Fortran 2003 provides intrinsics to do the latter

Also, in order to use C for specialised I/O
This is how MPI etc. are implemented
• I do NOT advise calling GUI libraries this way
Why Interoperate with C? (2)

C and C++ often need to call Fortran

Fortran has a wider range of faster numeric libraries
This is not just for historical reasons

Array handling in C and C++ is painful
It is often easier and runs faster using Fortran
• Especially true if you need to use OpenMP

That is why LAPACK etc. are often in Fortran
Merging Applications

Building a single program out of two or more
Where they are in a mixture of languages
Also calling a major library from another language

E.g. HPC code calling GUI libraries
In general, using modern Fortran and C++

- Strongly advise you to avoid doing this
  Always tricky — and can be fiendish

I am not going to describe the problems that arise
Multi-Program Applications

Better to build a multi-process application

http://people.ds.cam.ac.uk/nmm1/MultiApplics/

- May need to write special I/O functions
  But that is generally easier (see above)!

Recommended for using GUI interfaces in HPC
This lecture is a gross over-simplification. The area has always been diabolically complicated.

- This maps a safe path through the minefield. There is a huge amount more that it doesn’t mention.

- The languages have incompatible concepts. And implementations have a zillion variants. Also operating system variants, especially linker.

And it doesn’t even mention more than the basics.
A Quiz

How are these **implemented**? Are you **sure**?

```fortran
float fred ( char c , float f , int i [ 5 ] );
char joe ( ) ;

FUNCTION Alf ( a , b , c , d )
    COMPLEX :: Alf , a
    INTEGER ( INTENT = IN ) :: b ( 3 ) , c
    CHARACTER ( LEN = 52 ) :: d
END FUNCTION Alf
```

Don’t stop after the first 2–3 answers :-)  
No, I am **NOT** joking – so program **defensively**
Fortran to C Interoperability

Fortran standard is unexpectedly restrictive
Most of its restrictions are to enable portable coding

It is easily misinterpretable by C programmers
Regrettably, that means by most compiler developers

Important Note:
These are not mainly due to the design of Fortran
More the C standard and operating systems

• The main reason this lecture says what it does
  Write defensive code and you will rarely have trouble
The C Standard

- Using two C compilers has similar problems. Implementation variations in C alone are incredible. Can hit them using libraries, even the system’s ones.

- You will rarely do so under Linux on x86 etc. Every C vendor aims for gcc compatibility.

C is often said to be a simple language.
- That is not true, and has not been for 20+ years. The reasons and problems are subtle and arcane.

And C++ is 3–5 times as complicated as Fortran.
Basic Model

It defines Fortran kinds that map to C types

There is an intrinsic module to define the names

```
USE , INTRINSIC :: ISO_C_BINDING
```

And a BIND(C) attribute to specify C linkage

- Without it, Fortran does NOT define linkage
Arguments are not always passed as addresses
Derived types are not always laid out as written

Fortran allows much more optimisation than C/C++
Use of the Module

Compilers are allowed to define extra names
Also, future versions of the standards will do so
So it is strongly advised to use ONLY after USE

USE, INTRINSIC :: ISO_C_BINDING, &
ONLY C_PTR

Remember IMPORT for use in interface bodies

• As usual, I won’t do that in these slides
Any omission in specimen answers is a bug
Recommended Data Types

Fortran | C
---|---
CHARACTER(KIND=C_CHAR) | char
⇒ Note that char implies LEN=1
INTEGER(KIND=C_INT) | int
INTEGER(KIND=C_LONG) | long
REAL(KIND=C_DOUBLE) | double
TYPE(C_PTR) | void *

The most useful, and the safest
Compilation error if no match (KIND is -1)
Many others, but all have subtle gotchas
Fortran Default Types

- Currently, these are **NOT interoperable**.

  In practice, the following equivalences hold:

<table>
<thead>
<tr>
<th>Fortran Default Type</th>
<th>Interop. KIND</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>C_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>INTEGER</td>
<td>C_INT</td>
<td>int</td>
</tr>
<tr>
<td>REAL</td>
<td>C_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>REAL(KIND(0.0D0))</td>
<td>C_DOUBLE</td>
<td>double</td>
</tr>
</tbody>
</table>

Can pass them to C using **C_LOC** and **C_PTR**. Complex also carries across, where relevant.
Function Result Types

Can be only **scalars**, because of C constraints

Only **four types** are really safe, unfortunately

- In C terms, *int*, *long*, *double* and *void*

I.e. **C_INT, C_LONG, C_DOUBLE** and **C_PTR**

Can return **char, _Complex** and **derived types**

⇒ But don’t bet on them actually working :-(

Same applies to *float* and C’s zoo of **integer types**

- Due to compiler bugs, and may be **temporary**

The reasons have **nothing** to do with Fortran
Simple C Functions

Must use an explicit interface with BIND(C)

```fortran
FUNCTION Joe ( ) BIND ( C )
    REAL ( KIND = C_INT ) :: Joe
END SUBROUTINEINE Joe

PRINT *, Joe ( )
```

Can be used to call a C external function
Note the name is converted to lower-case for C

```c
int joe ( void ) { . . . }
```
High-Precision Timestamp

Returns the current time to **microsecond precision**
Just like `MPI_Wtime`, but more **general**

```c
/* Return high-precision timestamp. */
#include <stddef.h>
#include <sys/time.h>
double gettime ( void ) {
  struct timeval timer;
  if ( gettimeofday( & timer, NULL ) )
    return -1.0;
  return timer.tv_sec +
    1.0e-6 * timer.tv_usec;
}
```
Using the Timestamp

PROGRAM Timer
    USE , INTRINSIC :: ISO_C_BINDING , &
    ONLY : C_DOUBLE

    INTERFACE
    FUNCTION Gettime ( ) BIND ( C )
        IMPORT :: C_DOUBLE
        REAL ( KIND = C_DOUBLE ) Gettime
    END FUNCTION Gettime
    END INTERFACE

    REAL ( KIND = KIND(0.0D0) ) :: stamp
    stamp = Gettime ( ) ! This converts the KIND
    CALL Calculation
    PRINT * , "Time taken: " , Gettime ( ) – stamp

END PROGRAM Timer
Arguments

Normally passed as pointers to the first element
Applies to both scalars and arrays

Only explicit size and assumed size arrays
No assumed shape, ALLOCATABLE or POINTER
And CHARACTER must have LEN=1 (or default)

• But association lets you pass those types
What you can’t do is to use them in the interface

Procedure arguments are not allowed (but see later)
Interoperable Procedures (1)

Subroutines correspond to void functions

```fortran
INTERFACE
  SUBROUTINE Fred ( A, B ) BIND ( C )
    IMPORT :: C_INT
    INTEGER ( KIND = C_INT ) :: A, B
  END SUBROUTINE Fred
END INTERFACE

void fred ( int * p, int * q ) {
  . . .
}
```
Interoperable Procedures (2)

In exactly the same way, the C prototype:

```c
void fred ( int * , int * ) ;
```

Can call the Fortran external procedure:

```fortran
SUBROUTINE Fred ( A , B ) BIND ( C )
   INTEGER ( KIND = C_INT ) :: A , B
END SUBROUTINE Fred
```
Interoperable Procedures (3)

You can name the C function you call:

```fortran
SUBROUTINE John ( ) BIND ( C , NAME = 'Doe' )
END SUBROUTINE John
```

Interoperates with:

```fortran
void Doe ( ) ;
```

Note that this form does not lower case the string
Can also use when the C name is invalid in Fortran
INTENT(IN) and const

C const is not the same as INTENT(IN)
But, for pointer arguments, it is similar in purpose
You are recommended to match interfaces like this

SUBROUTINE Pete ( A , B , C ) BIND ( C )
   REAL ( KIND = C_DOUBLE ) , INTENT ( IN ) :: A
   INTEGER ( KIND = C_INT ) , INTENT ( IN ) :: B
   REAL ( KIND = C_DOUBLE ) :: C
END SUBROUTINE Pete

void pete ( const double * a , const int * b ,
           double * c ) ;
The above has used only facilities in Fortran 2003

TS 29113 extends it for arguments of procedures

Arguments can be assumed shape, assumed rank,
ALLOCATABLE, POINTER,
and assumed length CHARACTER

C interfaces provided to access such types

No change to return types or external data
nor procedure pointer arguments and variables
I hope to improve this area in the next standard
Other Procedures

In **some** cases, omitting the `BIND(C)` will work
But only in **some** cases, and only with **some** compilers

It is **not recommended** and **not portable**
But here is an **old course** that describes it

http://people.ds.cam.ac.uk/nmm1/MixedLang/

- **If possible**, convert to the **standard** mechanism
Arrays (1)

In general, arrays must be explicit shape
And their shapes must match in Fortran and C
Remember that array order is the other way round

```
INTEGER ( KIND = C_INT ) :: A ( 42, 221, 13 )
```

Corresponds to:

```
int a [ 13 ] [ 221 ] [ 42 ];
```

**Sequence association** relaxes this in some contexts
Treats that as a **vector** of length \(42 \times 221 \times 13\)
Arrays (2)

In arguments, they may also be assumed size

\[
\text{INTEGER ( KIND = C\_INT ) :: A ( 31 , 100 , * )}
\]

Corresponds to:

\[
\begin{align*}
\text{int a [ ] [ 100 ] [ 31 ] ;} \\
\text{int ( * a ) [ 100 ] [ 31 ] ;}
\end{align*}
\]

And, when used in appropriate ways only:

\[
\begin{align*}
\text{int a [ ] ;} \\
\text{int * a ;} \\
\text{int a [ ] [ 155 ] ;} \\
\text{int a [ ] [ 5 ] [ 2 ] [ 31 ] [ 2 ] ;}
\end{align*}
\]
Unfortunately, only LEN=1 is fully interoperable

The length is very like a first dimension
And remember the rules of sequence association

SUBROUTINE Fred ( N, A ) BIND ( C )
    INTEGER ( KIND = C_INT ) :: N
    CHARACTER ( KIND = C_CHAR ) :: A ( N )
END SUBROUTINE Fred

CHARACTER ( KIND = C_CHAR , LEN = 72 ) :: A ( 100 )
CALL Fred ( 72 , A ) ! This will work
CHARACTER (2)

C strings are null-terminated – Fortran’s are not

Remember char[4] is needed to store "123"
When moving to Fortran allow strlen()+1 bytes

You may need to add a null character when calling C
There is a C_CHAR constant C_NULL_CHAR for this
Also C_NEW_LINE and the other C escapes
All defined in the module ISO_C_BINDING

Alternatively, pass the length explicitly, as MPI does
VALUE Arguments

Puts value directly into the C argument list

- Other than one use, I advise avoiding VALUE
  Generally best to write C interface functions yourself
  Pass all arguments as pointers and convert if needed

C argument passing is far trickier than it seems
High chance of a Fortran compiler getting it wrong
Problems of both function results and derived types

- It can be done, but tricky to get reliably portable
  Again, this has nothing to do with Fortran
Anonymous Pointers (1)

TYPE(C_PTR) is equivalent of C void *
C_PTR can be assigned, used as arguments
Can even be used as the result type of functions

C_LOC intrinsic gets an address as a C_PTR
Needs either TARGET or POINTER attribute
Strictly, this example needs TS 29113, but works now

```
TYPE(C_PTR) :: ptr
INTEGER , TARGET :: array ( 1000 )
ptr = C_LOC ( array )
```
Anonymous Pointers (2)

With VALUE, can pass address of most variables

USE , INTRINSIC :: ISO_C_BINDING , &
ONLY : C_INT , C_PTR , C_LOC
INTERFACE
  SUBROUTINE Weeble ( n , a ) BIND ( C )
    IMPORT :: C_INT , C_PTR
    INTEGER ( KIND = C_INT ) , INTENT ( IN ) :: n
    TYPE ( C_PTR ) , VALUE :: a
  END SUBROUTINE Weeble
END INTERFACE
REAL , TARGET :: array ( 1000 ) ! No BIND(C)
CALL Weeble ( 1000 , C_LOC ( array ) )

void weeble ( int * n , void * b );
Anonymous Pointers (3)

A null pointer constant called \texttt{C\_NULL\_PTR}

- Recommended for initialising \texttt{C\_PTR}

\texttt{C\_PTR} does not initialise automatically

Test for null or identical using \texttt{C\_ASSOCIATED}

\begin{verbatim}
TYPE(C\_PTR) :: ptr1, ptr2, ptr3
ptr1 = function ( 1 )
ptr2 = function ( 2 )
IF ( C\_ASSOCIATED ( ptr1 ) ) . . . ! Non-null
IF ( C\_ASSOCIATED ( ptr1, ptr2 ) ) . . . ! Identical
IF ( C\_ASSOCIATED ( ptr3 ) ) . . . ! Undefined (error)
\end{verbatim}
Horrible Warning

- It is an error if the objects merely overlap
- Or if either argument doesn’t have a valid value
- Including when it has been deallocated

⇒ This applies in C, too – did you know?

    INTEGER (KIND = C_INT) , POINTER :: array (: )
    TYPE(C_PTR) :: ptr1 , ptr2
    IF ( C_ASSOCIATED ( ptr1 ) ) . . .       ! Undefined (error)
    ALLOCATE ( array ( 1000 ) )
    ptr1 = C_LOC ( array )
    ptr2 = C_LOC ( array (: 500 ) )
    IF ( C_ASSOCIATED ( ptr1 , ptr2 ) ) . . . ! Undefined (error)
    DEALLOCATE ( array )
    IF ( C_ASSOCIATED ( ptr2 ) ) . . .       ! Undefined (error)
Anonymous Pointers (4)

Can associate a Fortran pointer with a C_PTR value
If it is an array, you must also specify its shape

- Be warned – you get no type checking
The equivalent of casting void * to a typed pointer

```fortran
TYPE(C_PTR) :: ptr1, ptr2
REAL (KIND = KIND ( 0.0D0 ) ) , POINTER :: &
  scalar, array (:, :, :)
CALL C_F_POINTER ( ptr1 , scalar )
CALL C_F_POINTER ( ptr1 , array , &
  (/ 42 , 13 , 131 /) )
```
Derived Types (1)

Simple cases map onto C structures
C++ PODs are the idea – Plain Old Data

Only interoperable component types
No ALLOCATABLE or POINTER components

Derived types allowed as components, as in C
None of the more advanced properties
  None have been covered in this course

Explicit shape arrays are allowed, just as in C
Remember that array order is the other way round
Derived Types (2)

Unfortunately, **C struct layout is a can of worms**
Theoretically, the **Fortran** and **C** compilers match
**In practice**, that’s **far** too optimistic
The problems are far **too complicated** to describe

- **KISS** – i.e. make it **easy** for the **compiler**

Put **larger base types** before **smaller ones**
  E.g. **double** before **int** before **char**
Will **maximise** the chance of **reliable portability**
Will **usually** maximise the code’s **efficiency**, too
Example

TYPE, BIND ( C ) :: Packrat
    REAL ( KIND = C_DOUBLE ) :: array ( 40, 15 )
    INTEGER ( KIND = C_INT ) :: code
    CHARACTER ( KIND = C_CHAR ) :: message ( 72 + 1 )
END TYPE Packrat

typedef struct {
    double array [ 15 ] [ 40 ] ;
    int code ;
    char message [ 72 + 1 ] ;
}
External Data (1)

Variables in modules can be accessed from C. Any with BIND(C) map to an external variable. Ones without it do not create an external name.

```
MODULE Conglomerate
    USE, INTRINSIC :: ISO_C_BINDING
    INTEGER, ALLOCATABLE :: array (:, :)
    REAL ( KIND = C_DOUBLE ), BIND ( C ) :: visible
END MODULE Conglomerate
```

_visible can be accessed from C by:

`extern double visible ;`
External Data (2)

You can name the external variable, as before.
You can initialise it in either Fortran or C.
But you mustn’t do that in both, of course.

```
MODULE Whatever
    INTEGER ( KIND = C_INT ), &
    BIND ( C, NAME = 'Fred_3' ) :: x
    INTEGER ( KIND = C_INT ), BIND ( C ) :: PDQ = 456
END MODULE Whatever

extern int Fred_3 = 987;
extern int pdq;
```
Complex Numbers

Fortran interoperates with `C99 _Complex`
Sadly, `C99 _Complex` is horribly misdesigned
Few people use it – so WG14 has made it optional

- I don’t advise using it for function results
  Nor for arguments that use the `VALUE` attribute
  It will work with some compilers and not others
  You don’t want to know why, I can assure you

In practice, `C++ complex` has the same layout
But it is NOT fully compatible with `C99 _Complex`
I don’t advise these as result types or with VALUE.
Fine as pointer arguments or in external data.

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(KIND=C_SIGNED_CHAR)</td>
<td>signed char</td>
</tr>
<tr>
<td>INTEGER(KIND=C_SHORT)</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER(KIND=C_LONG_LONG)</td>
<td>long long</td>
</tr>
<tr>
<td>REAL(KIND=C_FLOAT)</td>
<td>float</td>
</tr>
<tr>
<td>COMPLEX(KIND=C_FLOAT)</td>
<td>complex float</td>
</tr>
<tr>
<td>COMPLEX(KIND=C_DOUBLE)</td>
<td>complex double</td>
</tr>
</tbody>
</table>
Other C Integer Types

You can pass unsigned integers as signed ones
- But stick to the values that are valid in both Fortran will always treat the values as signed

- Fortran has size_t but not ptrdiff_t
But size_t is an unsigned integer type!
- ptrdiff_t/size_t aren’t a signed/unsigned pair
But they will be in most implementations

C99 has a zoo of extended integer types
- Avoid them in interfaces – even in pure C
C specification is poor, and implementations differ
Procedure Pointers (1)

TYPE(C_FUNPTR) is an untyped procedure pointer
In C, all function pointers are compatible
I.e. they are different types, but with typeless values

- The procedure must be fully interoperable
  Not just BIND(C), but in C and C++, too
  ⇒ No inline, <stdarg.h> or C++ member functions

Use TYPE(C_FUNPTR), VALUE for arguments
You use C_FUNLOC just like C_LOC
Remember that C function type syntax is weird
Procedure Pointers (2)

There is a constant `C_NULL_FUNPTR` `C_ASSOCIATED` also works on `TYPE(C_FUNPTR)` `C_F_PROCPOINTER` converse of `C_FUNLOC`  

Procedure pointers and untyped values are both tricky  
⇒ Both together is doubleplus ungood (as in 1984)  
This will show the most trivial and safest uses  

`BIND(C)` internal procedures needs Fortran 2008  
Few compilers allow them yet, though `gfortran` does
This subroutine just calls its argument

SUBROUTINE Marshall ( arg ) BIND ( C )
  INTERFACE
    SUBROUTINE arg ( ) BIND ( C )
    END SUBROUTINE arg
  END INTERFACE
  CALL arg
END SUBROUTINE Marshall
The C equivalent of that subroutine is

```c
void marshall ( void ( * arg ) ( void ) ) {
    arg ( ) ;
}
```

Examples using internal and external procedures
Try them with both the Fortran and C marshall
PROGRAM McLuhan
  USE , INTRINSIC :: ISO_C_BINDING , &
  ONLY : C_FUNPTR , C_FUNLOC
  INTERFACE
  SUBROUTINE Marshall ( arg ) BIND (C)
    IMPORT :: C_FUNPTR
    TYPE ( C_FUNPTR ) , VALUE :: arg
  END SUBROUTINE Marshall
  END INTERFACE
  CALL Marshall ( C_FUNLOC ( Medium ) )
  CONTAINS
  SUBROUTINE Medium ( ) BIND ( C )
    PRINT * , "The medium is the message"
  END SUBROUTINE Medium
END PROGRAM McLuhan
PROGRAM McLuhan
  USE, INTRINSIC :: ISO_C_BINDING, &
  ONLY : C_FUNPTR, C_FUNLOC
INTERFACE
  SUBROUTINE Medium ( ) BIND ( C )
  END SUBROUTINE Medium
  SUBROUTINE Marshall ( arg ) BIND (C)
    IMPORT :: C_FUNPTR
    TYPE ( C_FUNPTR ), VALUE :: arg
  END SUBROUTINE Marshall
END INTERFACE
CALL Marshall ( C_FUNLOC ( Medium ) )
END PROGRAM McLuhan
SUBROUTINE Medium ( ) BIND ( C )
  PRINT *, "The medium is the message"
END SUBROUTINE Medium
C to Fortran

Try this with both the Fortran and C marshall

```c
#include <stdio.h>

extern void marshall ( void (*)( void ));

void Medium ( void ) {
    printf ( "The medium is the message\n" ) ;
}

int main ( void ) {
    marshall ( Medium ) ;
    return 0 ;
}
```
Practicalities

In theory, that’s all – but not in practice
The following has little to do with the standards

The most common areas I have seen cause trouble
• They are not a complete list of problem areas
Feedback on these guidelines would be appreciated

And remember rule number one:

• KISS – Keep It Simple and Stupid
Compatible Compilers

You need compatible Fortran and C compilers. Those from the same vendor usually are. E.g. gfortran and gcc or Intel ifort and icc. You can sometimes mix vendors, but not always.

- Use both in either 32- or 64-bit mode! Make sure the IEEE 754 modes are compatible. The same applies to some other compiler options.

- All this applies to C++ and C, incidentally.
Compilation and Linking

Compile all worker code without linking
• Link using compiler for master language

May need extra libraries, especially if C is master
Here is a way of find out which ones:

Usually option to display command expansion
   –v, –V, –#, –dryrun etc.
Link a dummy program using both compilers
Add any missing ones to (master) link command
GNU and Linux on Intel/AMD

Generally, the following will work:

```
gcc -c <other options> fred.c joe.c
gfortran <other options> alf.f90 bert.f90  
    fred.o joe.o
```

and:

```
gfortran -c <other options> alf.f90 bert.f90
gcc <other options> fred.c joe.c  
    alf.o bert.o -lgfortran -lm
```

You should put this in a Makefile, of course
Name Clashes (1)

Any external names in Fortran and C can clash Fortran external procedures, COMMON and modules, whether or not they have the BIND(C) attribute

Together with any C extern functions and variables
Remember extern is the default in file scope

- Avoid same name even when ignoring case
  Don’t use underscores at the beginning or end

Compilers vary a lot on name munging rules
It’s a bad idea to rely on that to protect your code
Name Clashes (2)

The really **nasty problems** occur with the **libraries**
All **C** library functions are all **external names**
And remember that **C++** includes the **C** library

Some variables, like **errno** and **math_errhandling**
Occasionally even **POSIX** ones, like **environ**

- Try to avoid **all** plausible external names
The **Fortran language** no longer has any
But **C** and **POSIX** do, and **Microsoft** may
Fortran and C++

Both of these can interoperate via C, in theory

• Unfortunately, C++ insists on being master
  Roughly corresponds to owning the main program
  May also involve owning the memory management

• Mixing them is very compiler-dependent
  Both need to be initialised and terminated properly
  Defined interfaces for this are now very rare

Many other issues, but most are mentioned later
Fortran is the Master (1)

• Generally, I recommend using this approach. The main exception is if you need to use C++

• Let’s start by assuming a Unix–like system. In this context, Microsoft and Macintosh are Unix–like.

Avoid using stdin, stdout and even stderr. stderr is the safest if you don’t use ERROR_UNIT.

• But it’s very compiler–dependent what works.

Opening other files using C or POSIX is OK.
Fortran is the Master (2)

Most of the C library works, including `<time.h>`

- `<stdlib.h>` is the main problem (but see later)
  Don’t expect `atexit()` etc. to work, though it may
  Occasionally used by a few libraries written in C
  Anything may happen if you call `exit()` etc.

  `malloc()` will work, if you don’t push it too hard
  `getenv()` and `system()` almost always work

- But what if the system isn’t Unix–like at all?
  Avoid `<stdio.h>` , `<stdlib.h>` , `<time.h>` or ask for help
C or C++ is the Master

Calling the Fortran 77 subset almost always safe
Fortran 90 facilities can be used with care

• Don’t use any of Fortran’s standard I/O units
  In rare cases, Fortran I/O won’t work at all

If you are very unlucky, ALLOCATE won’t work
That could also cause a few other things to fail

Call C to get at the program environment
For example, GET_COMMAND probably won’t work
I/O

Only the master will close files at termination
  • The worker must close its files explicitly
That’s generally good practice, even for the master

  • Use a unit/file from one language only
Never try to share stdin between languages
Best not to share stdout or stderr, either

The main problem is how to produce diagnostics
You can’t control ones from the run–time systems
Will often get mangled, and may even get lost
Shared Output

Can sometimes relax for stdout and stderr
Unix–like systems and GNU–like compilers only
Using stderr and ERROR_UNIT will often work

• Write complete lines and transfer immediately
  In Fortran use FLUSH after every transfer
  In C, use line buffering (setvbuf/_IOLBF) or flush()

• Never reposition or change any other I/O modes
  C++ cerr and stderr or ERROR_UNIT is risky

• Very compiler dependent and may fail horribly
Shared Memory Parallelism

- Use **threading** only in the **master language**
  Compile the **worker language** using **serial options**
  Remember that **threading** may call it in parallel

You can use a **threaded worker** from a **serial master**
It’s actually how **SMP libraries** are implemented
Doing that is **compiler-dependent** and for **experts only**

- Avoid **C++11 threading** – ask offline for why
  It’s not for use by **mere mortals** – I would have trouble

- Don’t **share I/O across threads/processes**
MPI and Distributed Memory

Each process runs separately, usually serially

- Using interoperability isn’t a problem
Signal Handling

- **Never** trap an *error* signal (*SIGFPE* etc.)
  And don’t even **think** of calling *raise* or *abort*
  You can trap a *non-error* signal, **set a flag** and **return**

```c
static volatile sig_atomic_t flag;

void handler ( int sig ) {
    flag = 1;
}

( void ) signal ( SIG_INT , handler );
```

- **Beyond that** **Beware of the Dragons**
Avoid like the Plague

- I strongly recommend not using C99 `<fenv.h>`
  Interacts horribly with both Fortran and C++ (sic)
  The Fortran modules `IEEE_...` are much saner
  But non-trivial use may cause C to misbehave

- Never return across a Fortran procedure
  I.e. A ⇒ Fortran B ⇒ C, and C jumps back to A
  Whether by `setjmp/longjmp` or C++ `throw/catch`

- And be very cautious when calling POSIX
  Far too complicated to describe what is safe