Fortran-C Interoperability

Hans Pabst
Application Engineer

Notices and Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

All products, computer systems, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Intel processors of the same SKU may vary in frequency or power as a result of natural variability in the production process.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

The benchmark results reported above may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804.

No computer system can be absolutely secure.

Intel® Advanced Vector Extensions (Intel® AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel® Hyper-Threading Technology available on select Intel® processors. Requires an Intel® HT Technology-enabled system. Your performance varies depending on the specific hardware and software you use. Learn more by visiting http://www.intel.com/info/hyperthreading.

Outline

- Subject and Problem Statement
- Traditional Techniques
- ISO_C_BINDING

- References

Subject and Problem Statement

Fortran-C* language interoperability enables reusing code in both directions.

How to make code interoperable?

- Some limitation to common constructs (true for ISO_C_BINDING and even more so for traditional techniques).
- Some familiarity required with both C and Fortran code.
- Use of tools to generate interfaces (out-of-scope here)

* Effectively any programming language which is C-interoperable.

Traditional Techniques

Traditional Techniques

… exploit that Fortran compilers

- … pass all procedure arguments by-reference (i.e. by-address).
- … usually represent optional arguments as NULL-pointers.
- … usually refer to functions and subroutines using lowercase un-mangled (C++: extern "C") names with trailing underscore (x → x_).
- … usually pass strings* with additional hidden arg. (by-value, at end of signature, determines length of string).
- … usually pass function return values via stack.

… work in both directions.

* Fortran **CHARACTER**-strings are not null-character terminated.

Traditional Techniques (cont.)

PROS*

- Can complement ISO_C_BINDING (additional compat. w/ F77)
- Shared impl. for trad. tech. and ISO_C binding possible
- Fosters robust API (due to limitations)

CONS

- Conditional compilation and dependency on compiler flags
- A-priori (implicit) knowledge e.g., of private data structures
- More limited compared to ISO_C_BINDING (lang. constructs)

* The reason "no other choice" is not an issue for the "PRO" category.

Example

The "greeting" subroutine (symbol name: greeting_) passes a character string and an integer number. A hidden argument (accumulated at the end of the function signature) denotes the length of the string (by-value). All non-hidden arguments are passed by-reference:

```
CALL greeting("Hello World!", 3) ! implicit/explicit interface is unrelated to interop.
```

```
#include <stdio.h>

void greeting_(const char* msg, const int* n, int len) {
	int i;
	for (i = 0; i < *n; ++i) {			/* repeated greetings */
		printf("%.*s\n", len, msg);		/* string is not 0-terminated */
	}
}
```

Recommendations

1. Prefer opaque handles with manipulation routines rather than exposing structured types, use create/destroy semantic if handle refers to dynamic resources (e.g., memory buffer).
2. Avoid to pass strings; if necessary use signature that writes into pre-allocated variable on the C or Fortran side (CHARACTER array of an "up-to" size) or yields C string literal (not dynamically allocated).
3. No "interoperable" functions i.e., do not exploit that return values are passed via stack; use subroutines w/ INTENT(OUT).

ISO_C_BINDING

Introduction and History

F2003: C interop. became an ISO standard

- Interop. is **explicit** using INTERFACE blocks with the BIND(C)
- ISO_C_BINDING is the name of a **module** related to standard; provided by compiler vendor (intrinsic)
- Theory: different modules per different/supported C compiler
- Notion of a "companion C compiler" often introduced earlier

ISO_C_BINDING module (content)

- Named constants (type-KINDs), derived types, helper routines

Built-in Types

Interoperable by using KIND (named constant)

- INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER e.g.,

```
INTEGER(C_INT)   :: i
REAL(C_DOUBLE) :: d
REAL(C_FLOAT)    :: f
```

- Unsigned integers are not introduced* to Fortran
  Special care must be taken if unsigned value-range is maximized out on the C-side.

- Unsupported mapping (value range or precision issue)
  - Value of KIND-constant is negative.
  - Decided by compiler vendor.

* Breaks typical assumptions/performance

FUNCTION or SUBROUTINE

Interoperable by using BIND(C)

- **Arguments** are passed by-reference (default), or by-value in case of VALUE attribute (without BIND(C), VALUE means something slightly different!)
  - POINTER or ALLOCATABLE attributes are not supported
  - Order in signature matches between C and Fortran
  - Variadic signature not supported
- **FUNCTION**: non-void function in C
  - Return value must be scalar
- **SUBROUTINE**: void function in C
  - No return value

BIND(C [, NAME="label"])

- Binding label determines the symbol name (linker)

* Unsigned integers are not introduced (breaks typical assumptions/performance)

SUBROUTINE* (Example)

```
INTERFACE
    SUBROUTINE greeting(msg, len) BIND(C, NAME="greeting") ! NAME is name of routine by default
        IMPORT C_CHAR, C_INT
        CHARACTER(C_CHAR), INTENT(IN) :: msg(*)
        INTEGER(C_INT), INTENT(IN), VALUE :: len
    END SUBROUTINE
END INTERFACE

CALL greeting("Hello World!"//C_NULL_CHAR, 3) ! string to be null-terminated or size needed on C-side
```

```
#include <stdio.h>

void greeting(const char* msg, int n) {
    int i;
    for (i = 0; i < n; ++i) {        /* repeated greetings */
        printf("%s\n", msg);         /* string is 0-terminated */
    }
}
```

* The "greeting" subroutine passes a character string and an integer number.

User-defined Types (UDTs)

- Derived data types in Fortran correspond to structured types in C ("struct")
  - Each element is an interoperable type (built-in or derived)
  - Element order and type-size (incl. arrays) must match
  - Element name not required to match between F and C
  - No ALLOCATABLE- or POINTER-components; code needs rework to rely on TYPE(C_PTR)
  - No bit-fields (no counterpart in Fortran)
- Unsupported
  - SEQUENCE and EXTENTS keyword (no counterpart in C)
  - Unions (no counterpart in Fortran)

- BIND(C) attribute ensures that the data layout (padding) matches data layout generated by "companion compiler".

User-defined Type (Example)

```fortran
TYPE, BIND(C) :: query
      INTEGER(C_INT) :: values(100), cmp
      INTEGER(C_INT) :: nvals
END TYPE

PURE FUNCTION ask(q) BIND(C)
      TYPE(query), INTENT(IN) :: q
      INTEGER(C_INT) :: ask
      ! count the number of matches
      ask = COUNT(q%cmp == &
            q%values(1:q%nvals)))
END FUNCTION
```

```c
typedef struct {
      int values[100], cmp;
      int nvals;
} query;

int ask(const query* q);
int main() {
      query q = {
            { 1, 2, 1, 4, 4, 3, 2, 3, 1 }, 1, 9
      };
      printf("-> %i\n", ask(&q));
      return 0;
}
```

* This example also shows how C is calling Fortran code.

Arrays*

- Built-in and derived/structured types supported
- Storage layout of multi-dimensional arrays
  - F: column-major (fast index first) e.g.,

    REAL(C_DOUBLE) :: a1(5), a2(6:7,18), a3(-7:8)
  - C: row-major arrays (fast index last) e.g.,

    double a1[5], a2[18][2], a3[16];

  In any case, declaring multiple ranks is just to let compiler generate the (linear) addresses when accessing the array.
- Non-zero rank sizes

* Special form of user-defined data type (UDT).

Storage Order

Row-major linear index from shape and multiple indexes:

```
size_t linear_index(const size_t index[], const size_t shape[], size_t ndims, size_t* size)
{
        size_t result = 0, size1 = 0;
        if (0 != ndims && NULL != shape) {
                size_t i;
                assert(NULL != index);
                result = index[0];
                size1 = shape[0];
                for (i = 1; i < ndims; ++i) {
                        result += index[i] * size1;
                        size1 *= shape[i];
                }
        }
        if (NULL != size) *size = size1;
        return result;
}
```

→ Column-major: *index* must be enumerated in reverse order.

Character Strings

- Character arrays are like normal array
  - Array size may be given separately
  - Do not need to be null-terminated

- Strings (in contrast to arrays)
  - Fortran code must null-terminate strings passed* to C code
  - ISO_C_BINDING module provides named characters e.g., C_NULL_CHAR or C_NEW_LINE

* No hidden size-argument generated for strings (FUNCTION or SUBROUTINE)

Lifetime of Variables

- Global data
  Fortran's SAVE attrib. is what's "static" in C.

  C: static keyword for variables is not only valid at global scope (e.g., "local statics"), variables at global scope (outside of any scope) are implicitly static even without the keyword.

  Fortran: SAVE is valid in a module or COMMON block, and
  BIND(C [, NAME="label"])
  is required for interoperability, BIND implies SAVE, BIND-label can (re-) name linker symbol.

- Thread-local storage (TLS)
  Not subject of ISO_C_BINDING
  but portable based on OpenMP:
  !$OMP THREADPRIVATE(*variable*)

```
MODULE mymod
    USE, INTRINSIC :: ISO_C_BINDING
    IMPLICIT NONE
    INTEGER(C_DOUBLE), BIND(C) :: array(8,8)
    INTEGER(C_INT), BIND(C, NAME="an") :: am
    INTEGER(C_INT), BIND(C, NAME="am") :: an
END MODULE
```

```
int array[8][8], am, an;
int main() {
    int i, j;
    for (i = 0; i < am; ++i) {
        for (j = 0; j < an; ++j)
            printf("%i ", array[i][j]);
        printf("\n");
    }
    return 0;
}
```

* Note: *am* and *an* are swizzled (row/col-major).

* C-code compacted to fit.

Lifetime of Variables (Example)

```fortran
MODULE mymod
	USE, INTRINSIC :: ISO_C_BINDING
	IMPLICIT NONE

	INTEGER(C_DOUBLE), BIND(C) :: array(8,8)
	INTEGER(C_INT), BIND(C, NAME="an") :: am
	INTEGER(C_INT), BIND(C, NAME="am") :: an

CONTAINS
	SUBROUTINE init() BIND(C)
		INTEGER :: i, j
		am = 4		!SIZE(array, 1)
		an = 2		!SIZE(array, 2)
		DO CONCURRENT(i = 1:am, j = 1:an)
		array(i,j) = i + j
		END DO
	END SUBROUTINE
END MODULE
```

```c
#include <stdio.h>

/*extern*/int array[8][8], am, an;

void init(void);

int main()
{
	int i, j;
	init();
	printf("%ix%i array:\n", am, an);
	for (i = 0; i < am; ++i) {
		for (j = 0; j < an; ++j)
		printf("%i ", array[i][j]);
		printf("\n");
	}
	return 0;
}
```

Pointers

Somewhat different concepts in Fortran and C

Fortran: typed memory address

- POINTER (to array) refers to array descriptor (with shape), and hence address calculation (index) is smart about array layout.
- Alias-analysis can be limited to (explicit) TARGETs.

C: (typed) memory address

- Pointer arithmetic is not aware of array dimensions, and type-casting (punning) can reach any (invalid) location.

Therefore, POINTER is not "reused" for C interoperability!

ISO_C_BINDING: derived types to represent C pointers

- TYPE(C_PTR) for pointers to interoperable data types
- TYPE(C_FUNPTR) for function pointers

Pointers (cont.)

ISO_C_BINDING: helper routines/functions

Conversion* (from Fortran, to C pointer)

- C_F_POINTER :TYPE(C_PTR) → POINTER
- C_F_PROCPOINTER: TYPE(C_FUNPTR) → PROC. POINTER

Address-of operators (get rid of Fortran descriptor; raw data)

- C_LOC(data), C_FUNLOC(procedure)

Comparison:

- C_ASSOCIATED(pointer) True if pointer is NULL
- C_ASSOCIATED(ptr1, ptr2) True if not NULL, and equal

* Adds descriptor/shape information in case of conversion to Fortran array-POINTER.

C_F_POINTER (Example)

```fortran
TYPE, BIND(C) :: query_type
      TYPE(C_PTR) :: values
      INTEGER(C_INT) :: cmp, nvals
END TYPE

FUNCTION ask(q) BIND(C)
      TYPE(query_type), INTENT(IN) :: q
      INTEGER(C_INT), POINTER :: v(:)
      INTEGER(C_INT) :: ask
      CALL C_F_POINTER( &
            q%values, v, (/q%nvals/))
      ask = COUNT(q%cmp == v)
END FUNCTION
```

```c
typedef struct {  /* number of values can be  */
      int* values;  /* unknown at compile-time */
      int cmp, nvals;
} query_type;

int ask(const query_type* q);
int main() {
      int v[] = { 1, 2, 1, 4, 4, 3, 2, 3, 1 };
      query_type q;
      q.values = v;
      q.nvals = 9;
      q.cmp = 1;
      printf("-> %i\n", ask(&q));
      return 0;
}
```

* FUNCTION is impure because of C_F_POINTER.

C_F_PROCPOINTER (Example)

```
ABSTRACT INTERFACE
    PURE FUNCTION func_type(i) BIND(C)
        IMPORT C_INT
        INTEGER(C_INT), INTENT(IN), VALUE :: i
        INTEGER(C_INT) :: func_type
    END FUNCTION
END INTERFACE

SUBROUTINE get_query(i, query)
    PROCEDURE(func_type), POINTER, INTENT(OUT) :: func
    INTEGER(C_INT), INTENT(IN) :: i
    INTERFACE
      PURE FUNCTION get_cfunc(i) &
      BIND(C, NAME="get_func")
        IMPORT C_INT, C_FUNPTR
        INTEGER(C_INT), INTENT(IN), VALUE :: i
        TYPE(C_FUNPTR) :: get_cfunc
      END FUNCTION
    END INTERFACE
    CALL C_F_PROCPOINTER(get_cfunc(i), func)
END SUBROUTINE
```

```
typedef int (*func_type)(const query_type* i);
query_type get_query(int i);

int ask(const query* q);
int trivial(const query* q) {
    return 0;
}

query_type get_query(int i) {
    switch (i) {
        case 0:     return trivial;
        case 1:     return ask;
        default:    return NULL;
    }
}
```

```
PROGRAM
    PROCEDURE(func_type), POINTER :: myfun
    CALL get_query(1, myfun)
    WRITE(*,"(A,I0)") "myfun(42) = ", myfun(42)
END PROGRAM
```

C_F_PROCPOINTER (Example II)

```
ABSTRACT INTERFACE
    PURE FUNCTION func_type(i) BIND(C)
        IMPORT C_INT
        INTEGER(C_INT), INTENT(IN), VALUE :: i
        INTEGER(C_INT) :: func_type
    END FUNCTION
END INTERFACE

TYPE :: functor_type
    PROCEDURE(func_type), &
        POINTER, NOPASS :: f
    TYPE(query_type) :: q
END TYPE

INTEGER(C_INT) :: values(:)
TYPE(functor_type) :: f
TYPE(query_type) :: q

values = (/ 1, 2, 1, 4, 4, 3, 2, 3, 1 /)
q%values = C_LOC(values)
q%nvals = SIZE(values)
q%cmp = 1

CALL get_query(1,q, f)
WRITE(*,*) f%f(q)
```

```
SUBROUTINE get_query(i, query, functor)
    TYPE(functor_type), INTENT(OUT) :: functor
    TYPE(query_type), INTENT(IN) :: query

    INTERFACE
      PURE FUNCTION get_cfunc(i) &
      BIND(C, NAME="get_func")
        IMPORT C_INT, C_FUNPTR
        INTEGER(C_INT), INTENT(IN), VALUE :: i
        TYPE(C_FUNPTR) :: get_cfunc
      END FUNCTION
    END INTERFACE

    CALL C_F_PROCPOINTER(get_cfunc(i), functor%f)
    functor%q = query
END SUBROUTINE
```

Recommendations

1. Prefer opaque handles with manipulation routines rather than exposing complicated structured types.
2. Avoid to hand-over lifetime of dynamic resources (e.g., memory buffer), or follow create/destroy semantic*.
3. Stream I/O avoids record delimiters that otherwise (binary I/O) must be parsed in C.

* Resource is released where it was created (either on C or Fortran side).

Dark Corners

- The "C companion compiler" (or theoretically multiple ISO_C_BINDING modules per different/supported C compiler) suggest that Fortran compiled objects (and module files) are not (link-) compatible between different Fortran compilers.

References

Acknowledgements

- Steve Lionel a.k.a. "Dr. Fortran" (Intel retiree)
- Martyn Corden (Intel)

References

- Language Interoperability (CSC Finland), Mikko Byckling, slides
- Interoperability with C in Fortran 2003, Megan Damon, slides
- Intel Fortran Compiler (18.0 Developer Guide and Reference): Standard Fortran and C Interoperability.
- GNU Fortran Compiler: Interoperability with C and Further Interoperability of Fortran with C.

intel®