DRAFT 0.1
Tips and Recommendations on Using Fortran in Conjuction with C++

C++ Languages and Tools Working Group
Run 2 Computing Strategy Committee

October 1, 1996

Contents

This document will provide practical tips on how to combine Fortran code into C++ projects, and how to incorporate C++ modules into mostly-Fortran code structures. It is intended for use by Run 2 programmers, particularly from CDF and D0. Though these experiments are committing to C++, their are several good reasons for knowing how to integrate with Fortran:

- There will be a good deal of legacy code, in Fortran 77 (with Vax extensions), and neither the resuorces nor any compelling reasons to re-write some fraction of it.
- There exist commercial and public domain libraries of useful routines, written in Fortran rather than C or C++.
- In some (assumedly rare) instances, new code may be written in Fortran, either because Fortran is partuiciularly well suited to efficient performance of some operation, or because an existing expert in some technique does not have the time to achieve a comfort level with C++.

These tips should allow people to integrate those Fortran modules with C++ code in a manner which is correct, consistent, and to some extent self-documenting. We point out some hidden traps and dangerous practices to avoid. And we address some of the subtle or difficult issues which are must be dealt with if the resulting code is to be robust.

A word about platform differences: The primary intent is to have code which does not have to change wehen moving amongst various supported platforms. Because this working group is *not* trying to define a "supported Fortran" version, and because some of these tips depend on details of the C++ or linker, that ideal will not always be practical. We will at least try to point out instances where a tip will not work in the usual manner on some platforms/compiler combinations.

Along those lines: There are some resources available, in the form of files which help smooth over the inter-platform differences. Fermilab will see to it that stable "reference" copies of those files are available locally. This document will discuss how such files can be used to allow for cross-platform development while avoiding the readability problems that can stem from lots of `#ifdefs` in application code.

We divide the tips into three types: The basics of combining the two languages; more advanced techniques to handle less standard situations; and warnings about things which are dangerous or are highly non-portable.

Before addressing any of these tips, we will define a framework of includes and macros, implemented via a preprocessor, which lets the user do the basics in a clean and platform-independant way. The C++ Working Group (and/or its successors) will supply and maintain these. In a sense, the remainder of the document supplies the "Why" behind this framework, as well as being guidelines for how these macros and utilities are built for various platforms.

1 Basic Framework

1.1 Fundamentals

The overall program structure will be a C++ mainline, with C++ fucntions calling, when necessary, Fortran-coded functions and subroutines. The steps the user takes to support the Fortran calls are:

1. The user creates an special Fortran declaration file—which will be pre-processed to create an include file—allowing the system to know about Fortran routines that may be called. The include file derived from the Fortran declaration file needs to be included each C++ modules that calls one of those Fortran routines. We suggest the Fortran declaration file have extension "`.hf`"; the include file created will have extension "`.h`". For purposes of illustration, we will call these `legacyF.hf` and `legacyF.h`.

2. The name of the preprocessor will be `cxxFhead`; this should be installed in a generally accessible bin area so that lines may appear in make files looking like:

   ```
   legacyF.h: legacyF.hf
     cxxFhead legacyF.hf > legacyF.h
   ```

3. The Fortran declaration file consists of lines each declaring one Fortran subroutine or function that the C++ code may call.

   The form and contents of these lines are described below:

4. Subroutine and function declarations must begin in column 1 and look like

   ```
   F77SUBROUTINE hxinvert ( REAL*8(5,5), REAL*8(5,5) )
   INTEGER F77FUNCTION countrax ( REAL*4, REAL*4(1000,14) )
   ```

   The structure of these lines should be clear from the above examples: The routine name (preceeded by the return type if it is a function), followed by a collection of argument types.

   All argument types in this declaration are FORTRAN types:

   ```
   INTEGER
   INTEGER*1    INTEGER*2    INTEGER*4    INTEGER*8
   LOGICAL      LOGICAL*1    LOGICAL*2    LOGICAL*4
   REAL         REAL*4       REAL*8
   CHARACTER*n
   ```

These keywords—but not the macros F77SUBROUTINE or F77FUNCTION—may be presented in upper or lower case.

5. Strings passed as a CHARACTER*n, where n is any length, will always be copied into a structure organized the way Fortran expects a string, truncated if necessary to n and copied back at the end—so make sure the C++ char* passed points to an area with enough room for n bytes, plus one for the terminating zero.

6. Arrays may also be passed:

   ```
   INTEGER(5)
   REAL*8(20,15)
   ```

   These dimensions may be variables which must then contain valid values by the time the Fortran routine is invoked. In particular, if some limit is expressed as a constant or #define *which will be valid by the time* legacyF.h *is included* then an array dimension can use this limit.

   The last dimension, which varies slowest in Fortran, may be 1 in the declaration and yet be more than one when used.

7. The user is responsible for supplying the array in the order Fortran will expect. Fortran arrays are kept "in a different index order" than in C or C++. More accurately, C does not support the idea of multi-dimensional arrays, and the most basic syntax that can be treated as an array is a vector of vectors of vectors ...such as x[5][8][4]. When this construct is used, the last index varies most rapidly, just the oppposite of Fortran if you wish to treat x[k][j][i] as meaning the Fortran x(i,j,k). Also, Fortran array indices by defalt start a1 1, not 0. However, in C++ the user can define a more Fortran-like syntax for arrays (that same array could be set up for use with syntax x(i,j,k) mapping to the data member x[k-1][j-1][i-1]). At any rate, this basic framework does not provide for "automagic" hidden copying of arrays into the Fortran ordering.

8. Sometimes the declaration of a Fortran routine will take more characters than could fit on a single line. The structure rules for the Fortran declaration file are:

   - Each declaration must start in column 1; lines which are indented are treated as continuations of the previous declaration.
   - C++-style comments (initiated by // and extending to the end of the line) are accepted and ignored.
   - Blank lines are permitted and ignored.

   Thus a suggested style for declarations might be:

   ```
   INTEGER F77FUNCTION listtrax // returns number of tracks
     REAL*4 (1000,14),          // array of hit positions
     REAL*4 TRAX(1000,50) )     // list of up to 50 tracks found
   ```

9. Any compilation unit (a compilation unit is a code file along with its entire tree of includes) in C++ which may wish to call one of the Fortran routines declared in the above manner, must `#include` the `legacyF.h` file before calling any of the Fortran routines. The generated `legacyF.h` can be included in multiple compilation units to be linked together (or for that matter in multiple times in the same file) without problems or conflicts. It will not be necessary to explicitly include `f77toCxx.h` in the C++ code file that is using the Fortran subroutine calls.

10. When a function in C++ needs to call one of the Fortran routines declared in the above manner, it uses the routine name supplied in the declaration. This name is case sensitive. The user need supply no other declaration of the function he is calling; the "function prototype" is generated and appears in `legacyF.h`.

    The function call needs to use the correct data types for the arguments. The C++ language provides a couple of nice fetaures in this regard: Although the declaration will be set up to expect "pass by reference", C++ automatically converts arguments appropriately, so that you can just use the variable name. Yet if the wrong type of data is passed (an integer where a real is expected) this error will be caught by the compiler instead of producing misbehavior at runtime. The macros provided will even result in compile-time checking for the right *number* of dimensions in an array argument, though not for a match between the size expected and the size provided in each dimension.

    For example:

```
// legacyF.h
  #include <f77toCxx.h>
  F77FUNCTION ( INTEGER, countrax, REAL*4(1000,14) )
  F77SUBROUTINE ( hxinvert, REAL*8(5,5), REAL*8(5,5) )
  // ...
// mycode.cc
  #include "legacyF.h"
  int i;
  float   trax[14][1000];
  double dtrax[14][1000];
  float   trix[14000];
  float   oops[13][1000];
  //...
  i=countrax (&trax); // OK - actual declaration expects a reference
  i=countrax ( trax); // OK - C++ promotes to expected reference
  i=COUNTRAX ( trax); // ERROR - name of routine is case sensitive
  i=countrax (dtrax); // ERROR - double dtrax is NOT REAL*4
  i=countrax ( trix); // ERROR - CONTRAX expects a float** (2-D arrray)
  i=countrax ( oops); // This goof won't be caught at compile time
  //...
```

11. Fortran routines involving arguments which are structures (other than arrays or character strings) can be dealt with but are outside this basic framework—see §3.1.

1.2 Additional Options within Basic Framework

The basic framework allow flexibility beyond the above fundamental features for supporting Fortran calls:

- The user-written legacyF.hf contains information which allows expansion into proper C++ declarations and necessary wrappers. This expansion may be different for different platforms, but the preprocessor will know about this via environment variables. In fact, both the user C++ code that calls the Fortran routines and the legacyF.hf file should be completely platform independant.

  The requirements for interfacing to Fortran might, in principal, depend on which C++ compiler is being used. However, the same generated legacyF.h will work for all C++ compilers we know about and sanction on any given platform—it will contain tests of environment variables and conditionals that will make this work.

- An argument in the declaration for a Fortran routine may optionally be preceeded by the keyword CONSTANT. This promises that the Fortran routine will not change the data of that argument; it therefore allows C++ to feel comfortable passing a variable which was declared to be const to that routine.

  ```
  // in legacyF.h:
  INTEGER F77FUNCTION countrax ( CONSTANT REAL*4(1000,14) )
  F77SUBROUTINE hxinvert ( CONSTANT REAL*8(5,5), REAL*8(5,5) )
  // in C++ code:
  const double hx[5][5];
  double hy[5][5];
  //...
  hxinvert ( hx, hy); // OK -- first argument expects a const
  hxinvert ( hy, hy); // OK -- can pass a non-const instead
  hxinvert ( hy, hx); // ERROR caught by compiler -- hxinvert might
                      // change the value of the const hx.
  ```

  Remember, however, that while you have promised that the Fortran routine won't change the value of its argument, nothing in Fortran prevents you from breaking that promise (with likely bad consequences). We have used CONSTANT rather than const to emphasize that this is not the C++ enforced version of constancy!

- Multiple Fortran declaration files (specifying distinct sets of Fortran routines!) may be created. In that case, each one is to be preprocessed into the corresponding include file, and one or more such include files may be included in a given C++ compilation unit. If the same Fortran routine is declared in two places, this conflict will be caught by the C++ compiler.

- There will be a file f77toCxx.h defining supporting macros for declaring Fortran routines to be called from C++ and so forth. Note that this name is kept short for Windows convenience, and that we will assume below that the file is installed in a system include area.

  The generated legacyF.h file will itself include the file defining the supporting macros:

  ```
  include <f77toCxx.h>
  ```

  Therefore, even if the user wishes to directly use definitions in f77toCxx.h, that file need not be included explicitly in any file which inludes the generated legacyF.h.

  The version of f77toCxx.h supplied may not be identical on all platforms; the installation process will ensure that the proper version is in place.

- While the issue of different C++ compilers on the same platform is dealt with as above, the question ofdiffernt Fortran compilers cannot be handled in the same way. Unless there is only one Fortran compiler on a platform, or they all share the same linkage conventions, the preprocessor has no way of knowing which compiler created a gived F77 routine being declared. In such circumstances, a default compiler will be assumed, but the user can specify a different one (that the preprocessor knows about) by adding the compiler name in front of the F77 declaration:

  ```
  INTEGER ABSOFT F77FUNCTION countrax
    ( CONSTANT REAL*4(1000,14) )
  MICROWAY F77SUBROUTINE hxinvert
    ( CONSTANT REAL*8(5,5), REAL*8(5,5) )
    // in C++ code:
  ```

  This mechanism is ugly; it spoils platform independance of the "`.hf`" file, and creates a trap if somebody recompiles a Fortran routine using a compiler with different linkage conventions. But this ugliness is unavoidable since there is no way of knowing at compile time what the linkage convention to a non-C++ routine might be. We suggest sticking to one default Fortran compiler—or a set of compilers with identical linkage conventions—for a given platform!

.2

We profusely thank, exalt and glorify **FILL IN NAME HERE** who has created the preprocessor, macros and includes that make this mechanism work across various platforms.

2 Basics of Integrating C++ and Fortran

The basics of combining C++ and Fortran depend on two pieces of knowledge:

- How symbol names are formed from variable and function names, to produce the names known to the linker. C++ "mangles" names in idiosyncratic ways, but it provides a means to specify incorporation of symbols which are not mangled. So we can use the knowledge of Fortran naming conventions to provide a means of calling Fortran routines from C++.

- How Fortran handles arguments and return values of functions and subroutines. C++ has the flexibility to handle a wide variety of conventions; fortunately, Fortran implementations use a well-defined particular set of conventions to which C++ can interface.

Some of these basic tips are absolutely essential; others provide a uniform framework for integrating Frotran routines into C++ programs. In either case, there is no reason to ignore or violate any applicable part of this basic guidance.

2.1 Structure of an Integrated Program

Obviously there is no single organization forced on all programs involving both C++ and Fortran. Nonetheless, we are going to proposed a fixed general structure, asserting that straying from this organization will likely lead to serious or at least bothersome problems.

- The main program is in C++. A C++ main program sets up a lot of things, and while it is conceivable that you could do these in the context of a Fortran main program, doing this in a portable way would involve a lot of work, it could be done at all. The C++ main program calls other C++ routines, in a variety of compilation units, as is normal.

- Some of these compilation units contain functions which need to call Fortran routines. Each such compilation unit has (via include files) declarations describing the calling of those routines and declaring that the routine names are external to the C++ naming conventions. If necessary, wrappers to do any interface work are also present; these wrappers may differ from platform to platform. The user code is likely to employ a standard, supplied means of automating the generation of these declarations and wrappers (§1).

- At each place where a Fortran routine needs to be called, it is invoked as if it were a C++ routine (at global socpe) with the declared name. Argument passing is done in the normal fashion, and return values (for a function) are done as if it were any other function call.

- The Fortran code can call other Fortran routines linked in, in the normal way. (There are ways to allow the Fortran routines to take advantage of C or C++ routines further down the call tree; this advanced technique is described in §3.4.) Eventually, the routine at the top of the Fortran tree will return, back to the calling C++ code.

2.2 Calling a Fortran Subroutine From C++

2.2.1 Subroutine Name

The name a C++ program will use to call a Fortran subroutine (or function) is the subroutine name demoted to lower case and, for most systems, with an underscore appended. (However, there is no uniformity in this convention—some Fortran compilers will generate names without the underscore, or in whatever case the Fortran code uses to declare the subroutine.)

For example, `SUBROUTINE CONDENSE` is known as `condense_`. We will refer to that as the "linker name"; it is needed when setting up a function prototype. As to the name your C++ code will use to call that subroutine, you can choose whatever name for it you want; we'll refer to that as the "chosen name". Unless there is some conflict, we recommend you choose the Fortran name, without the underscore, and consistently use all lower-case or all upper-case letters. In our example the chosen name would be `condense`.

In general, the tips here will work from the chosen name, creating a wrapper which ultimately calls the linker name to call the actual Fortran routine. The one subtlety occurs when the Fortran compiler on a platform will not append an underscore to form a link name, and the chosen name is identical to the linker name. In that case, one fewer level of wrapping is needed—the chosen name is declared (see §2.2.2) directly as an `extern "C"` routine. This is a headache primarily from the standpoint of creating clean automated schemes for generating the wrappers from a minimal amount of information.

2.2.2 Prototype in Header File

C++ requires a function prototype declaring the nature of the function, so it can tell what sort of arguments are to be passed, and check that the right sort of arguments were supplied. File organization styles tell you to put these prototypes into "header files" (normally with extension `.h`). Nominally there should be one header file for subroutine; if a group of Fortran subroutines are always grouped together, we see nothing wrong with combining the prototypes for all of them into a single header file. The header file is meant to be included (using `#include` in any C++ file in which one of those subroutines may be called.

The following mantra goes a bit beyond only creating a prototype, in that it also defines the C++ callable routine, to be a tiny wrapper calling the Fortran routine. We will see later (§2.8) that when certain types of arguments are involved, this wrapper can get a bit less trivial, but it is still suitable for keeping in the hearder file.

For this illustration, we assume the linker name of the subroutine will be `condense_` and that the chosen name is `condense`. In the file `condense.h` place the following lines: Check these

```
// Header file for Fortran 77 Subroutine condense(I, J, X)
#ifndef CONDENSE_H                                                // 1
#define CONDENSE_H                                                // 1

// CONDENSE takes two integer arguments and uses Wimpy's          // 2
// algorithm to set the value of the third, REAL*4 argument,
// to the Wimp function of I and J.

extern "C"
void condense_ ( const int& i, const int& j, float& x );          // 3

inline                                                            // 4
void condense ( const int& i, const int& j, float& x ) {          // 5
  condense_ ( i, j, x ) } ;                                       // 3


#endif                                                            // 1
```

An explanation of the above file—we have placed numbers at the end of each line corresponding to items in this explanation. (Actual header files should not include these numbers, although their presence would not cause any problems.)

1. Header files should always use this trick so they will work properly even if the user includes them multiple times.

2. It is highly desirable to put some explanation of what a subroutine does as a comment in the header file.

3. The `extern` keyword tells the compiler not to "mangle" the supplied name, so that the linker will look for the symbol `_condense`.

   Why not `extern "FORTRAN"`? Because only `extern "C"` is standard C++. It would be nice to be able say that `extern "FORTRAN" myname` would create whatever linkage is needed to call the Fortran routine "`myname`" but that would involve interacting with the linker—the C++ compiler knows nothing about which Fortran compiler you might want to involve! `extern "C" x` merely says this subroutine is to be known as "x", and not as some name incorporating additional information useful (for type checking) to C++.

The nature of how arguments are handled is discussed in the next few subsections (§2.3, §2.4, §2.7, §2.8).

4. A C++ program could directly call the linker name. We will be doing one extra level of indirection by using the chosen name instead. But by telling the compiler to inline the call, we incur zero extra code or execution time.

   The `inline` directive serves another purpose, partcularly relevant for automated generation of these wrappers. If `condense` were to be called from two separate C++ routines appearing in two distinct compilation units, then the wrapper function `condense` would be defined twice—once in each compilation unit. If both those definitions were of global scope, this would be an illegal conflict. The `inline` directive limits the scope of the definition to only one compilation unit (a `static` directive would serve this purpose equally well).

5. The line declaring `void condense` establishes the name a C++ routine can use to call the routine. By doing this, we can ensure that even if some platform has unusual FORTRAN symbol conventions (so that the name in lines 3 of the header would have to change) the C++ programs calling the FORTRAN routine can remain untouched.

2.3 Passing Subroutine Arguments

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

2.4 Integer and Real Data Types

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

2.5 Functions and Return Values

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

2.6 How the Subroutine Call Looks in C++

Once the header is set up in the above manner, the subroutine can be called just like any C++ routine. One elegant feature you should utilize is that although the passing convention is by reference (expecting the address of each argument) C++ knows what to do if you just pass the argument. So (in our example) although

```
#include condense.h
int m, n;
float x;
...
condense ( &m, &n, &x );
```

would work, so would the nicer-looking

```
#include condense.h
int m, n;
float x;
...
condense ( m, n, x );
```

You *could* also call the link name condense_ directly but you shouldn't because it may reduce portability.

2.7 Arrays

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

2.8 Character Strings

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

2.9 Linking Fortran with C++

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3 More Advanced Techniques

Here we present techniques to handle less standard situations, or to help support superior code design—appropriate exception handling, avoidance of redundancy in source code, and so forth.

3.1 Sharing Parameter and Structure Definitions

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.2 Variables in Named Common Blocks

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.3 Mixing Fortran and C++ I/O

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.3.1 "Printed" Output to Standard Output

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.3.2 C++ Access to Fortran-Created Files

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.3.3 Fortran Access to C++-Created Files

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.3.4 Input—Interactive and Text Files

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.4 Fortran Subroutines Calling C++ Routines

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.4.1 Dealing with Namespaces

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.5 Exception Handling

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.6 Posix System Calls From Fortran

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.7 Isolating Fortran Names in a Namespace

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

3.8 Optional and Default-valued Arguments

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

4 Things You Don't Want To Do

These are ways to do things which either are dangerous, are highly non-portable, or represent abysmal code practice. We make no representation that any example discussed below works on multiple platforms; the examples are illustrations of techniques which should be avoided.

4.1 Fortran Mainlines Incorporating C++ Routines

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

4.2 Un-named Common

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

4.3 Voluntary Aborts Directly From Fortran

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

4.3.1 What to Do Instead of CALL EXIT()

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

4.4 Establishment of Exception Handlers From Fortran

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

4.5 System Calls That Should Not Be Done From Fortran

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

5 Resources

Information in this document was culled from:

- Verballly communicated Expertise from various members of the C++ Working Group.
- The README, .doc file, and .h file of cfortran, a DESY product in the CERN libraries, available at ftpzebra.desy.de. This purports to automatically solve all problems in interfacing C and Fortran but falls short of that ambitious goal. Still, it is a good resource for finding the various machine-differing conventions.
- XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX