How to Interface Fortran to C++

BENNO LIST

4.10.1998

This note gives some recipies how to interface Fortran to C++, i. e. how to call Fortran routines from a C++ program. It is currently not complete or exhaustive, and some statements apply only to certain operating systems, mostly to HPUX as run on SnOPAL.

Comments, corrections etc. to
`Benn.List@cern.ch`.

1 Data Types

The basic data types are very similar in Fortran and C++. Tab. 1 is a table of these correspondences.

Although there is basically a one–to–one correspondance between various Fortran and C++ data types, this is a bit of a coincidence, i. e. this is not guaranteed by the language definitions to hold on all systems. Therefore it is recommended to use `typedef`s that for each Fortran data type define the corresponding C++ data type to make wrapper files easily portable. A header file named `FTypes.h` has been set up which contains such `typedef`s, and their

Fortran	C++	`FTypes.h`
`INTEGER*2`	`short`	`FInteger2`
`INTEGER`	`int`	`FInteger`
`INTEGER*8`	`long long`	`FInteger8`
`REAL`	`float`	`FReal`
`REAL*8`	`double`	`FReal8`
`REAL*16`	`long double`	`FReal16`
`LOGICAL*1`	`bool`	`FLogical1`
`LOGICAL*2`	`short`	`FLogical2`
`LOGICAL`	`int`	`FLogical`
`CHARACTER`	`char`	`FCharacter`
`CHARACTER*`n	`char[`n`]`	
`COMPLEX`		
`COMPLEX*8`		

Table 1: Correspondance of data types of Fortran and C++. The column `FTypes.h` gives the names of the `typedefs` defined in the header file `FTypes.h` for the Fortran data types.
`INTEGER*8` is not available on some machines, e. g. HPs.

names are also given in Tab. 1.

It is highly recommended to use these `typedef`s, because the C++ standards does *not* define the lengths of several data types, and the correspondances between FOPRTRAN and C++ data types may vary from machine to machine.

1.1 Arrays

Arrays work much the same way in Fortran as in C++, with the notable exception that array indices start at 1 in Fortran as a default (although one can define them to start at 0), and *always* start at 0 in C++.

For multidimensional arrays there is one difference: Arrays in Fortran are stored such that the first index "runs fastest", i.e.

```
      REAL X(3,2)
```

is stored in the order

`X(1,1), X(2,1), X(3,1), X(1,2), X(2,2), X(3,2),`

and therefore corresponds to

```
#include "FTypes.h"
FReal x[2][3];
```

which is stored in the order

`x[0][0], x[0][1], x[0][2], x[1][0], x[1][1], x[1][2].`

Therefore the Fortran-`X(3,1)` corresponds to the C++ element `x[0][2]`. Note also the obligatory use of *two* brackets! Unfortunately, writing `x[0,2]` is also legal (although a decent compiler should warn about it), because the comma–operator means "evaluate expressions from left to right, overall result is result of the rightmost expression." Therefore, `x[0,2]` correponds to `x[2]`, i. e. accesses the *third row* of `x`, which in this case means also that the subscript is out of range, which will generally not be detected by the compiler or even at runtime, unless it leads to a segmentation fault. So be careful!

2 Common Blocks

Common blocks are entities that are used in Fortran to exchange data between subroutines. They are somehow equivalent to global variables in C++, and global variables can (and must) be used to define common blocks. Note that all Fortran common block names have a lower–case spelling in C++, with a trailing underscore "_".

Examples:

```
C ... Size of HBOOK store
      PARAMETER (NWPAWC=100000)
C ... HBOOK common
      COMMON /PAWC/ H(NWPAWC)
```

in Fortran can be implemented as

```
#include "FTypes.h"
enum {NWPAWC =  100000};       // Size of HBOOK store
FReal pawc_ [NWPAWC];          // HBOOK common
```

in C++. Note that the linker "knows" only about the *common block* name, not about the names, types or sizes of the variables that make up the common block. The name of the common block corresponds just to the adress of its first element. Therefore it is necessary to have a variable in C++ with the name of the common block (remember, lower case plus underscore) that is located at the start of the common block. It is sensible to define a `struct` that has the structure of the common block, and its name, as in the next example:

```
      PARAMETER (MAXTRK = 100, MAXHIT = 300)
      COMMON /CMTRK/
     +       NTRACK, NHITS(MAXTRK), PX(MAXTRK), PY(MAXTRK),
     +       PZ(MAXTRK), XHITS(MAXHIT,MAXTRK), YHITS(MAXHIT,MAXTRK),
     +       ZHITS(MAXHIT,MAXTRK)
```

in Fortran can be translated to C++ as

```
enum {MAXTRK = 100};
enum {MAXHIT = 300};
struct CMTRKType {
     FInteger nTrack;
     FInteger nHits [MAXTRK];
     FReal px [MAXTRK];
     FReal py [MAXTRK];
     FReal pz [MAXTRK];
     FReal xHits [MAXTRK][MAXHIT];
     FReal yHits [MAXTRK][MAXHIT];
     FReal zHits [MAXTRK][MAXHIT];
};
CMTRKType  cmtrk_;
CMTRKType& cmtrk = cmtrk_;
```

The definition of the reference in the last line is not really necessary, but makes for a nicer syntax, because now one can access the elements through e. g.

```
// cmtrk definition as above goes here...

int main() {
     cmtrk.nTrack = 1;
     cmtrk.px[0] = 0;      // note use of "0" as index!
     cmtrk.py[0] = 0.5;
     cmtrk.pz[0] = 1.3;
     cmtrk.nhits[0] = 2;
```

```
    cmtrk.xhits[0][0] = 0;
    cmtrk.yhits[0][0] = 3.0;
    cmtrk.zhits[0][0] = 5.0;
    cmtrk.xhits[0][1] = 0;
    cmtrk.yhits[0][1] = 3.3;
    cmtrk.zhits[0][1] = 5.5;
    return 0;
}
```

3 Subroutine Calls

3.1 Call Without Parameters

First of all, a function prototype has to be defined for the Fortran subroutine. This prototype has to have "C" linkage; the name is the name of the Fortran routine, spelled lower case, plus a trailing underscore:

```
      SUBROUTINE BLABLA
      PRINT *, 'I''m a totally useless Fortran subroutine'
      RETURN
      END
```

in Fortran can be called from C++ with:

```
// best put in header file "blabla.h"
extern "C" {
    void blabla_();
}

int main() {
    blabla_();
    return 0;
}
```

A note on the trailing underscore: On some machines, especially HPs (i. e. SnOPAL, hpplus) the `f77` compiler needs to be told to produce the trailing underscore. On HPs, this is done by the "`+ppu`" compiler option. The use of the trailing underscore is OPAL standard on all machines, so as long as one uses the "`fort`" script to produce object files, the trailing underscore is always there.

3.2 Wrapper files

The last example can be made a bit prettier with the use of a "wrapper", like this:

```
// best put in header file "blabla.h"
extern "C" {
```

```
     void blabla_(); // original Fortran subroutine, not to be called
                     // by user directly
}

inline void blabla {blabla_();} // wrapper. inline definition leads
                                // to no runtime overhead for use
                                // of the wrapper.

int main() {
     blabla();      // replaced by call to blabla_ at compile time.
     return 0;
}
```

This use of wrappers becomes important when parameters are handed over.

3.3 Parameters

Generally, Fortran always uses call–by–reference, i. e. at a subroutine call *always* an adress is handed to the subroutine, *never* the data directly. This is even the case when a subroutine is called with a constant argument, e. g. for

```
      X = SIN(1.5)
```

a temporary variable is created by the compiler, filled with the value 1.5, and then `SIN` is called.

C (as opposed to C++), on the other hand, *always* uses call–by–value, i. e. it passes *always* the value of a variable to a subroutine, never its location. However, the location of a variable `x` is given by `&x`, so this can be used to transfer the required information. Therefore

```
      SUBROUTINE HLIMIT (NPAW)
      INTEGER NPAW
C ... Code here for whatever HLIMIT does...
      RETURN
```

must have a definition like this:

```
// best put in header file "hbook.h"
#include"FTypes.h"
extern "C" {
     void hlimit_ (FInteger *npaw);
}
```

and can be used like this:

```
#include"hbook.h"
#include "FTypes.h"
enum {NWPAWC =  100000};       // Size of HBOOK store
FReal pawc_ [NWPAWC];          // HBOOK common
```

```
int main() {
     FInteger nwpawc = NWPAWC; // Cannot take adress of NWPAWC, that's
                               // only a symbolic constant!
     hlimit_ (&nwpawc);        // initialize PAW
     return 0;
}
```

Of course, this is rather clumsy. This is where wrappers are really useful:

```
// wrapper for hlimit_
// best put in header file "hbook.h"

     inline void hlimit (int npawc) {hlimit_ (&npawc)}
```

Note that because `hlimit_` does not change the value of its argument, the wrapper uses call–by–value. Also note that `hlimit` uses and `int` argument instead of `FInteger`. `Finteger` is certainly only a `typedef` for `int`, but even if that were to change, C++ would make the necessary conversion, so all would be good. Now we can use `hlimit`:

```
#include"hbook.h"
#include "FTypes.h"
enum {NWPAWC =  100000};       // Size of HBOOK store
FReal pawc_ [NWPAWC];          // HBOOK common

int main() {
     hlimit (NWPAWC);          // initialize PAW
     return 0;
}
```

3.4 Parameters That Are Changed

As opposed to C, C++ has a construct for call–by–reference. Therefore, the Fortran routine

```
      SUBROUTINE DOIT (I)
      I = 2
      RETURN
      END
```

can be called from C++ with

```
#include "FTypes.h"
// best put into "doit.h"
extern "C" {
     void doit_ (FInteger *i);
}
     inline void doit (FInteger& i) {doit_ (&i);}
```

```
int main() {
     FInteger i = 5;
     doit_ (&i);
     i = 5;
     doit (i);
     doit(5);            // compiler error!
     int j = 5;
     doit (j);           // compiler error IF FInteger is not typedef
                         // for int
     float f = 5;        // f is now 5.0
     doit (f);           // compiler error!
}
```

Actually, one could also define

```
#include "FTypes.h"
// best put into "doit.h"
extern "C" {
     void doit_ (FInteger& i);
}
```

and use `doit_` the same way `doit` is used above, but the wrapper still makes things nicer in that the trailing underscore is not needed.

3.5 Passing of Strings

Strings are handeled rather differently in Fortran and C or C++. C uses arrays of characters for strings, where the last character is set to '
0' to indicate the end of the string.

When a string is passed to a Fortran subroutine, the subroutine expects the adress of the string, and the length of the string, which is given *after* all other arguments. The string length is passed *by value, not by reference*! If several strings are passed, the lengths of all strings are passed in the corresponding order:

Here is an example:

```
      SUBROUTINE HBOOK1 (ID, CHTITL, NX, XMI, XMA, VMX)
      INTEGER ID, NX
      CHARACTER*(*) CHTITL
      REAL XMI, XMA, VMX
C ... Code here
      RETURN
      END

      SUBROUTINE HRPUT (ID, CHFILE, CHOPT)
      INTEGER ID
      CHARACTER*(*) CHFILE, CHOPT
      RETURN
```

```
      END
```

will have a warpper file like this, where the length of the string is evaluated by the standard C function strlen:

```
// wrapper for hlimit_
// best put in header file "hbook.h"
#include <cstring>

extern "C" {
  void hbook1_ (FInteger *id,
                const FCharacter *chtitl,
                FInteger *nx,
                FReal *xmi,
                FReal *xma,
                FReal *vmx,
                FInteger chtitl_length);
  void hrput_ (FInteger *id,
               const FCharacter *chfile,
               const FCharacter *chopt,
               FInteger chfile_length,
               FInteger chopt_length);
}
inline void hbook1 (int id,
                    const FCharacter *chtitl,
                    int nx,
                    float xmi,
                    float xma,
                    float vmx) {
   hbook1_ (&id, chtitl, &nx, &xmi, &xma, &vmx, strlen (chtitl));
}
inline void hrput (int id,
                   const FCharacter *chfile,
                   const FCharacter *chopt) {
  hrput_ (&id, chfile, chopt, strlen (chfile), strlen (chopt));
}
```

and can be used like this:

```
void f() {
     hbook1 (100, "Histo 100", 100, 0, 1, 0);
     hrput (0, "histo.hbook", "N");
}
```

Note that the last three integer arguments of `hbook1` will be automatically converted to reals, as the user would expect (wrongly expect if he used Fortran!). Note also that the strings are declared as `const FCharacter *` in the header file. This tells the compiler that the subroutines do not change the contents of the strings (which may be a lie! The compiler has no means to check this).

4 Function Calls

5 Linking

C++ differs from FORTRAN, among other things, in that it provides constructors and destructors. C++ guarantess that for any global object (i. e. objects that are defined outside any function, imcluding the main routine) the constructor is called before the main program starts, and the destructor is called after the end of the main program. This is not foreseen in FORTRAN, and therefore any program that uses C++ routines must, in general, be linked with a C++ linker, and have a main program written in C++.

A new version of the "`link`" script has been developed for HPs and shift. It is called `clink''` and can be found on
`/afs/cern.ch/user/b/blist/.hp700_ux100/bin`
for HPs and
`/afs/cern.ch/user/b/blist/.sgi_64/bin`
for shift. To link a C++ main program with C++, just give the additional option `-C`, everything else works as usual.