

CPPF77

55-69 minutes

Carsten A. Arnholm

A portable technique for Windows NT/95, UNIX and other systems

Version 1.1, 28-May-1997

Contents

[1. INTRODUCTION](#)

[1.1 MOTIVATION](#)

[1.2 REQUIREMENTS FOR SUCCESSFUL RE-USE](#)

[1.3 MAIN PHILOSOPHY](#)

[2. FORTRAN FEATURES SUPPORTED](#)

[2.1 SUBROUTINE AND FUNCTION CALLS](#)

[2.1.1 Single value parameters](#)

[2.1.2 Array parameters](#)

[2.2 UNSUPPORTED OR UNTESTED FEATURES](#)

[3. IMPLEMENTING A FORTRAN INTERFACE IN C++](#)

[3.1 LINKAGE CONVENTIONS](#)

[3.2 CALLING CONVENTIONS](#)

[3.3 PROTOTYPING A SUBROUTINE IN C++](#)

[3.4 PROTOTYPING A FUNCTION IN C++](#)

[3.5 PASSING PARAMETERS FROM C++ TO F77 AND BACK](#)

[3.5.1 Passing single-value parameters](#)

[3.5.2 Passing single-dimension array parameters](#)

[3.5.3 Passing multi-dimension array parameters](#)

- [3.5.4 Passing single-value COMPLEX parameters](#)
 - [3.5.4.1 FORTRAN functions returning COMPLEX values](#)
- [3.5.5 Passing single-value CHARACTER parameters](#)
 - [3.5.5.1 Solution strategy](#)
 - [3.5.5.2 single CHARACTER string example](#)
 - [3.5.5.3 Passing single-dimension CHARACTER array parameters](#)
- [3.6 LINKING C++ AND FORTRAN](#)
- [4. ACHIEVING PORTABILITY](#)

- [4.1 THE IDEA OF STUB CODE GENERATION](#)
- [4.2 HEADER FILES CONTAINING FORTRAN PROTOTYPE DECLARATIONS](#)
- [4.3 AUTOMATIC STUB CODE GENERATION BASED ON PROTOTYPE HEADER FILES](#)
 - [4.3.1 The HCOMP utility program](#)
- [5. THE FORTRAN.H FILE](#)

- [5.1 THE COMPLEX CLASS](#)
- [5.2 THE CHARACTER CLASS](#)
- [5.3 THE FMATRIX CLASS](#)
- [6. TUTORIAL EXAMPLES](#)

- [6.1 CHARACTER EXAMPLE](#)
 - [6.1.1 How to call a function returning a CHARACTER](#)
- [6.2 COMPLEX EXAMPLE](#)
 - [6.2.1 How to call a function returning a COMPLEX](#)
- [6.3 FMATRIX, A 2-DIMENSIONAL ARRAY EXAMPLE](#)
- [6.4 MIXED LANGUAGE FILE I/O EXAMPLE](#)

This document has been created based on my practical experience from more than 10 years of professional FORTRAN 77 (from now on referred to as F77) software development and maintenance, and about 4 years of similar C++ experience. With such a background, I found it natural to try to mix the two languages by calling FORTRAN from C++.

I was surprised to learn that there was no standard way of calling F77 code from C++. To my knowledge, all relevant de-facto standards interface F77 and C, not C++. I

consider C and C++ to be separate languages, and some of the features in C++ are better suited for seamless integration with F77 than what is available in C. Especially, the ability to pass function parameters by reference and the ability to express F77 types as classes makes C++ superior to C for seamless integration with F77.

For these reasons, I decided to develop and document such a standard. The hope is that it will be perceived as general enough to serve as a de-facto standard for portable, mixed C++/F77 programming. If that happens, the document has served its purpose well.

If you have any comments or questions relating to the contents of this document, feel free to contact me by e-mail at ca@dnv.com or Carsten.Arnholm@dnv.com.

1.1 Motivation

Re-use is a popular buzzword among C++ designers and programmers, and usually the topic discussed is: "How do I write software to make it re-usable ?". Another, much more practical question relating to re-use is: "How do I re-use software that has already been written ?". The last question is inspired by one of Bjarne Stroustrup's statements: "to be re-usable, software must first be usable" [Stroustrup,1993]. This document is addressing the last question, since existing software, with all its flaws, has one major advantage over yet-to-be-written code: It has been proven to be usable.

FORTRAN has been one of the most popular computer languages used for science and engineering for almost 40 years (the first FORTRAN language versions emerged in the mid-1950's). Much of the software developed in this period is now irrelevant and forgotten, but a large portion remains in use. In addition, new software is still being written in FORTRAN, simply because it is a proven and well established technology among scientists and engineers. Through the language's history of standardisation it has also become easily portable across operating systems, which is fundamental for software that often needs to run on many kinds of hardware, as well as live through several generations of computing trends.

Since the 1950's, the software industry has changed and grown to become one of the dominating industries, and the complexity of software programs has grown in a similar manner. Today, almost all engineering software programs deal much more with general information management than fundamental numeric computation. Even though the FORTRAN language has developed and improved dramatically through the '66, '77 and '90 standards, it has failed to adopt the technique which today is

recognised by many as the key to mastering complexity: Object Orientation.

For these reasons, scientists and engineers are turning more and more towards languages that support object orientation (OO), and especially C++ which is by far the largest and most popular OO language (ignoring for a moment the latest Java hype). Still, it would be too much of a revolution (and no good idea) to throw away all the FORTRAN software which has been tested and proven useful, especially if it can be shown that some of it (not all) actually fits quite well within the new OO domain.

The right question to ask is obviously: How can existing FORTRAN code be 'plugged' into new OO programs written in C++ ? The following sections will set the premises for this question and attempt to answer it to a level which can serve as a de-facto standard for creating portable mixed language programs written in both C++ and FORTRAN.

1.2 Requirements for successful re-use

When writing mixed language programs, one must make sure it is done in a way which does not add new constraints or increase complexity, as compared to writing the programs in one language only. Adding new constraints may drastically reduce the life-time of the software, while increasing complexity will have a negative effect on development and maintenance costs (thereby possibly cancelling the intended cost saving when re-using existing code).

These general requirements can be expressed in more concrete ways:

- The C++/F77 programs must be as portable as their FORTRAN-only ancestors.
- A single source code must be used on all platforms.
- Calling F77 code from C++ must be easy and straightforward. It must not be significantly more difficult to call FORTRAN from C++, compared to calling it from FORTRAN itself.
- Mixed C++/F77 code must not induce any significant performance penalty.
- All major F77 features must be supported from C++.
- Calling F77 from C++ shall be done without changing the F77 code, which has been tested and verified.

1.3 Main philosophy

The main idea of interfacing C++ and FORTRAN presented in this document is based on the SUBROUTINE and FUNCTION language elements of F77. Other language elements, like common blocks, are not viewed as suitable for interfacing directly within C++. See also [section 2.2](#) for further details.

As mentioned in section 1, several FORTRAN standards exist. This document addresses interfacing C++ and F77, since most of the software that is relevant for re-use has been written according to a standard complying with F77. More specifically, the new features of Fortran 90 (MODULEs, user defined TYPEs, user defined REAL precision, labelled subroutine parameters, etc.) are not addressed.

It is, however, possible to use the techniques described in this document to create mixed C++ and Fortran 90 programs, as long as an F77 subset is used in the interface between the two languages.

2.1 SUBROUTINE and FUNCTION calls

In F77, the definitions given below are typical, and may serve as basic examples of the kind of subroutine and functions that are easily called directly from C++. As can be seen from the examples, the F77 standard is followed strictly (notably uppercase code and no more than six characters in names). Some of these archaic rules can be relaxed as seen from a C++ point of view, but adhering to a well known standard is usually a very good idea, especially when portability issues come into play.

In the first example, a subroutine SUB1 takes a LOGICAL as input, and returns a CHARACTER string as output:

```
SUBROUTINE SUB1( FIRST, NAME )
  LOGICAL          FIRST
  CHARACTER*(*)    NAME
  IF( FIRST ) THEN
    NAME = 'Elin'
  ELSE
    NAME = 'Arnholm'
  ENDIF
  RETURN
END
```

Second, an INTEGER function IFUNC1 that takes no input parameters and returns a

value read from a common block:

```
INTEGER FUNCTION IFUNC1
INTEGER          IVALUE
COMMON /STORE/  IVALUE
IFUNC1 = IVALUE
RETURN
END
```

Third, a REAL function RFUNC1 that takes a REAL as input and multiplies it with some value before returning it as the function value:

```
REAL FUNCTION RFUNC1(RVALUE)
REAL          RVALUE
REAL          PI
PARAMETER (PI=3.1415926)
RFUNC1 = 2.0*PI*RVALUE
RETURN
END
```

Fourth, a REAL function RFUNC2 that takes two INTEGERS as input which are multiplied with each other. The result of the multiplication is converted to REAL type using a FORTRAN intrinsic function, and subsequently returned as the function value.

```
FUNCTION RFUNC2( IVALUE, JVALUE )
INTEGER          IVALUE, JVALUE
RFUNC2 = REAL( IVALUE*JVALUE )
RETURN
END
```

Note on implicit types in F77: RFUNC2 is a REAL function because the return type is not specified, and the name of the function does not begin with letter I,J,K,L,M or N, which would have made it an INTEGER function. The same rule applies to any variable or function that has no explicit type definition. Needless to say, this way of programming should be avoided in the future, but a lot of FORTRAN software exist which use implicit types, and a C++ programmer therefore needs to understand it.

2.1.1 Single value parameters

The previous section gave several examples of FORTRAN parameter types in

routine/function calls. Formally, the ANSI F77 datatypes are:

- INTEGER
- REAL
- DOUBLE PRECISION
- COMPLEX
- LOGICAL
- CHARACTER [*n], where n is the optional string length (in the range 1 to 32767)

All of the above datatypes, except COMPLEX and CHARACTER, have a direct counterpart in basic C++ types which makes it easy to express parameters to F77 subroutines.

The COMPLEX type is used for complex arithmetic, where real and imaginary terms are involved. A COMPLEX may be viewed as a "struct" containing 2 REALs, the first representing the real term, and the second representing the imaginary term. The COMPLEX type can therefore be expressed using a C++ struct or class (as long as no virtual functions are involved), and passed directly to F77 functions. Certain restrictions apply for FUNCTIONs returning COMPLEX results as function value. This is described in [section 3.5.4.1](#).

The CHARACTER type is a more fundamental special case due to the different ways of handling string lengths in F77 and C++, as well as the different ways of passing strings as function parameters. The solution to this problem is described in a [section 3.5.5](#). Note also that FUNCTIONs returning CHARACTER results as function value are currently not automatically supported (see [section 6.1.1](#)).

2.1.2 Array parameters

F77 supports arrays with up to 7 dimensions. Data is always stored in a contiguous block of memory, and there is a standard organisation of data within that contiguous block of memory. For a one-dimensional array there is only one possible way (in practice) of organising data, and this matches what is done in C++. Passing single-dimension arrays between F77 and C++ is therefore straightforward (see [section 3.5.2](#) for a practical example).

For two-dimensional (and higher dimensional) arrays, F77 uses a "column-first"

convention which is opposite to the C++ convention, which could be termed "row-first". The reader should note that this is another special case of non- conformance between F77 and C++ that must be addressed specifically. The solution to this problem is described in [section 3.5.3](#), and a complete tutorial is available in [section 6](#).

2.2 Unsupported or untested features

F77 has some features that are less often used, and which are either not supported within this context, or the author has made little or no attempt at confirming whether the feature can be interfaced directly from C++. When attempting to re-use F77 code using these features, a general recommendation is to write new wrapper routines in F77 with interfaces that conform to the subset of supported features.

The unsupported or untested features can be summarised as follows, in an approximate order of decreasing importance:

- COMPLEX FUNCTIONs are not automatically supported, see restrictions in [section 3.5.4.1](#), and example in [section 6.2.1](#).
 - CHARACTER FUNCTIONs are not automatically supported. See also [section 6.1.1](#).
 - Non-standard type expressions, such as REAL*16 etc. are not supported.
 - Accessing FORTRAN common blocks directly from C++ can be done, but portability has not been tested. An alternative is to write F77 access functions instead.
 - The ENTRY facility in FUNCTION/SUBROUTINE calls has not been tested.
 - Alternate returns in FUNCTION/SUBROUTINE calls are not supported, as FORTRAN statement labels have no meaning in C++.
 - Functions as subroutine parameters has not been tested. Be aware of potential problems if attempting to provide C/C++ functions as parameters to F77 routines. Recommendation: stay away from such use, or write a wrapper F77 subroutine and pass that subroutine instead.
-

3.1 Linkage conventions

In C++, several functions may share the same name, as long as the parameter types are not identical. This feature is called function overloading. Function overloading is commonly implemented by use of "name mangling", i.e. the parameter types of the function parameters become part of the function name, as seen from the perspective

of the compiler.

FORTRAN does not allow function overloading, and consequently name mangling is not performed by F77 compilers. In order for C++ compilers to recognise code generated by an F77 compiler, name mangling must be switched off for these routines. By using the SUBROUTINE and FUNCTION macros as specified in sections [3.3](#) and [3.4](#) below, name mangling will be properly switched off. This is because both the SUBROUTINE and the FUNCTION macros include the definition `extern "C"`.

3.2 Calling conventions

Another area of difference between C++ and FORTRAN is the calling convention, i.e. how are parameters pushed on the call stack, and who is responsible for tidying up the stack after a function has been called, the calling or the called function? Also, the questions of "name-decoration" (i.e. leading and/or trailing underscores in combination with function names), and case sensitivity belong in this discussion.

This is an area where things are inherently platform dependent, but generally one might say that C++ follows the `__cdecl` calling convention (allowing among other things a variable number of arguments in a function call), while FORTRAN follows the `__stdcall` calling convention, which is generally incompatible with `__cdecl`. In C++, it is easy to call a function using a non-default calling convention. You can simply specify it in the prototype declaration.

Again, using the SUBROUTINE and FUNCTION macros as specified in sections [3.3](#) and [3.4](#) below, will ensure that the proper calling convention is being used.

3.3 Prototyping a SUBROUTINE in C++

By including the header file `fortran.h`, you will gain access to all the declarations and macros required to declare C++ prototypes representing F77 subroutines. More specifically, the SUBROUTINE macro may be used when prototyping an F77 subroutine. Below is an example of how a subroutine taking no parameters is prototyped in C++:

```
#include <fortran.h>
SUBROUTINE F77SUB( );
```

3.4 Prototyping a FUNCTION in C++

In F77, a FUNCTION is exactly the same as a SUBROUTINE, except that it returns a function result (the result variable is returned by value, contrary to subroutine and function parameters, which in F77 are passed by reference). Similar to the SUBROUTINE macro, several macros are defined for the purpose of prototyping F77 functions returning different data types:

Macro name	Type of value returned by F77 function
INTEGER_FUNCTION	INTEGER
REAL_FUNCTION	REAL
LOGICAL_FUNCTION	LOGICAL
DOUBLE_PRECISION_FUNCTION	DOUBLE PRECISION

Below is an example of how an INTEGER FUNCTION taking no parameters is prototyped in C++:

```
#include <fortran.h>
INTEGER_FUNCTION F77FUN( );
```

3.5 Passing parameters from C++ to F77 and back

3.5.1 Passing single-value parameters

Simple typedefs are provided for declaring parameters to be passed to F77 routines. For most 32 bit systems the examples given below will work, but since both 16 bit (DOS/WIN3.1) and 64 bit (DEC/ALPHA) systems are in use today, these typedefs may be redefined when moving to such platforms. To achieve portability, it is therefore good practice not to use the native C++ types directly when calling F77 routines.

```
typedef int    INTEGER;
typedef float  REAL;
typedef double DOUBLE_PRECISION;
typedef int    LOGICAL;
```

Note that in function prototypes, these emulated F77 type names must be specified with a trailing ampersand (&, that is) when declaring simple parameters (not arrays). The ampersand character is used to indicate "pass by reference" which is always used in standard F77. Note that it is not possible to pass normal variables "by value" which is the default in C/C++.

Some FORTRAN compilers do provide facilities for passing parameters by value, but

this requires changing the existing FORTRAN code to become less portable and standardised, and it also violates the principle of re-using the FORTRAN code without touching it. Such facilities should therefore not be used.

As an example, the F77 function RFUNC1 from [section 2.1](#) is properly prototyped and used from C++:

```
#include <fortran.h>
REAL_FUNCTION RFUNC1(REAL& RVALUE);

double cppfunc(double& value)
{
    REAL RVALUE=(REAL) value;
    REAL RETVAL = RFUNC1(RVALUE);
    return (double)RETVAL;
}
```

3.5.2 Passing single-dimension array parameters

Passing single-dimension arrays is as easy as passing single-value parameters. The only difference is that array parameters in F77 routines must be prototyped with a trailing asterisk in C++ prototypes, indicating that a pointer is to be passed. The C++ array name is then simply passed as a parameter in the call.

Note that a common source of confusion is the fact that F77 arrays by default start with index=1 (although this can be user-defined), while a C++ array always start at index=0. The user should therefore be careful not to confuse indexing in the two languages. The next example illustrates how an array is passed. First the F77 subroutine (LARR is the array length):

```
      SUBROUTINE F77SUB( LARR,ARRAY )
      INTEGER          LARR
      REAL              ARRAY( LARR )
      INTEGER I

      DO 1000 I=1,LARR
          ARRAY(I) = REAL( I*I )
1000  CONTINUE
      RETURN
```

END

C++ function calling the F77 subroutine:

```
#include <fortran.h>
SUBROUTINE F77SUB( INTEGER& LARR,
                  REAL* ARRAY);
void cppfunc()
{
    const INTEGER size=10;
    REAL    ARRAY[size];
    INTEGER LARR=size;
    .
    F77SUB( LARR,ARRAY);
    .
    .
    .
    .
    return;
}
```

3.5.3 Passing multi-dimension array parameters

Consider a case where the following SUBROUTINE returning a two-dimensional array must be called from C++ (LARR1 & LARR2 are the array dimensions):

```
        SUBROUTINE F77SUB( LARR1,LARR2,ARRAY)
        INTEGER          LARR1,LARR2
        REAL             ARRAY( LARR1,LARR2 )
        INTEGER I,J

        DO 2000 J=1,LARR2
            DO 1000 I=1,LARR1
                ARRAY(I,J) = REAL(J*1000 + I)
1000      CONTINUE
2000    CONTINUE
        RETURN
        END
```

Passing a multi-dimensional array as exemplified above is slightly more complicated than passing a single-dimension array, and can be a source of error and inefficiency. As mentioned earlier, the main problem is that F77 and C++ have inherently incompatible array representations (data ordering is different).

To solve this problem with a minimal run-time overhead, the user should consider which of the two following cases apply in each particular case:

- Case 1. The array is constructed locally, and used only for calling FORTRAN functions (i.e. nowhere is the C++ notation `array[i][j]` in use). This can be handled by providing a matrix class (FMATRIX class, [section 5.3](#)) that is compatible with F77 arrays, and which supports its own subscripting notation.
- Case 2. The array is constructed or used in other C++ functions, and is not limited to use by FORTRAN. This requires explicit data conversion due to the incompatible conventions (see also [section 6.3](#) for a complete tutorial on how this can be safely achieved using the FMATRIX class).

Case 1 multi-dim. array example (array used only locally: conversion avoided):

```
#include <fortran.h>
SUBROUTINE F77SUB( INTEGER& LARR1,INTEGER& LARR2,REAL* ARRAY);
void cppfunc()
{
    INTEGER          LARR1=3,LARR2=2;
    FMATRIX<REAL>    ARRAY( LARR1,LARR2 );
    size_t           index1,index2;

    F77SUB( LARR1,LARR2,ARRAY);

    index1 = 2;
    index2 = 1;
    float value = (float) ARRAY(index1,index2);
}
```

Case 2 multi-dim. array example (array used elsewhere: must use conversion):

```
#include <fortran.h>
SUBROUTINE F77SUB(INTEGER& LARR1,INTEGER& LARR2,REAL* ARRAY);
void      cppsub2(float array[3][2]);
void cppfunc()
{
    const      INTEGER size1=3,size2=2;
    REAL      array[size1][size2];
    INTEGER    LARR1=size1,LARR2=size2;
    size_t     index1,index2;

    {

        FMATRIX<REAL>  ARRAY(&array[0][0],size1,size2);
        F77SUB(LARR1,LARR2,ARRAY);

    }

    index1 = 2;
    index2 = 1;
    float value = (float) array[index1][index2];
```

```

        .
        cppsub2(array);
        .

    return;
}

```

3.5.4 Passing single-value COMPLEX parameters

As mentioned earlier, the COMPLEX type is used for complex arithmetic, where real and imaginary terms are involved. The real and imaginary terms are each represented by a single REAL or DOUBLE PRECISION value. The F77 COMPLEX type does not have a native counterpart in C++ (although the new C++ Standard Library has), but one can easily be implemented as a C++ class template. [Section 5.1](#) describes the COMPLEX class template implementation in detail.

The basic principles of passing a COMPLEX variable to F77 are outlined below. First, a skeleton F77 routine is presented, it takes a single COMPLEX parameter.

```

SUBROUTINE F77CPX(CPXVAL)
COMPLEX          CPXVAL
.
RETURN
END

```

Second, to call this routine from C++, one would generally do:

```

#include <fortran.h>
SUBROUTINE F77CPX(COMPLEX<REAL>& CPXVAL);
void cppfunc(float& rval, float& ival)
{
    COMPLEX<REAL> CPXVAL(rval,ival);
    F77CPX(CPXVAL);

    rval = CPXVAL.real();
    ival = CPXVAL.imag();
}

```

3.5.4.1 FORTRAN functions returning COMPLEX values

These functions must be treated as exceptions, due to the peculiar way some FORTRAN compilers return COMPLEX function values. This means that extra manual work is forced upon the programmer, compared to functions returning other variable types. There are two solutions to choose from here:

1. Write a new F77 wrapper subroutine

Call the COMPLEX FUNCTION from the new F77 subroutine with an extra call parameter representing the function value. Call the new F77 subroutine from C++ instead of the original function.

1. Drawbacks

1. Double function call overhead

2. Unnatural call syntax

2. Advantages

1. Portable on all platforms without any code modifications

2. Write a new C++ wrapper function

Call the F77 COMPLEX FUNCTION from the new C++ function in a way which is compatible with your FORTRAN compiler. This new wrapper function takes the same parameters as its F77 relative, and it returns a function value of type COMPLEX<REAL>. [See section 5.1](#) for a full description of the template class COMPLEX<class T>.

1. Drawbacks

1. C++ wrapper function must be re-implemented for different FORTRAN compilers

2. Advantages

1. No extra function call overhead if C++ wrapper is inlined

2. Call syntax to wrapper function is identical to original F77 function

In both cases, the new subroutine or function must be given a new unique name, compared to the original F77 function. The second solution is also illustrated in detail in [section 6](#).

3.5.5 Passing single-value CHARACTER parameters

A simple typedef is not sufficient to describe a C++ type which would be compatible with the CHARACTER type available in F77, for two reasons:

First, a basic character string is in C++ represented via the `char*` type. The length of the string is determined by the position of the zero-termination character `'\0'`. In F77, the string is never zero-terminated. Instead, an integer value representing the declared length of the CHARACTER accompanies the string itself, and the length value is always available via the F77 intrinsic function `LEN(string)`.

Second, the F77 standard does not specify the implementation of passing CHARACTER strings to subroutines and functions. Consequently, several incompatible implementations are used in different FORTRAN compilers. These differences can be exemplified using C++ terminology:

- Some compilers pass a struct containing a `char*` pointer and a `size_t` value for each CHARACTER string passed.
- Other compilers pass a `char*` pointer for each CHARACTER string, and then pass a `size_t` value for each CHARACTER string *as a series of hidden parameters at the end of the parameter list*.

Obviously, if one used either of the above possible passing methods directly (i.e. used the method required by the FORTRAN compiler at hand), portability would be lost. The code would also look ugly and inelegant, and it would be very easy to make mistakes.

3.5.5.1 Solution strategy

To cope with the language differences, simplify passing of CHARACTER strings, and still maintain portability, the CHARACTER must be implemented as a class in C++. The requirements to the CHARACTER class may be summarised as:

1. The CHARACTER class must be able to use an existing `char*` pointer so that both C++ and F77 have a common string representation. This is necessary to maintain performance, and allow F77 to return strings to C++ without requiring string copy functions to be called after returning to C++.
2. The CHARACTER class must be able to construct a correct string length value recognised by F77, so that the intrinsic `LEN(string)` function as well as space padding behaves properly. I.e. standard F77 behaviour must be allowed:
 1. F77 truncates assigned strings when the CHARACTER variable is too short
 2. F77 pads the CHARACTER variable with blanks when assigned string is short

3. The CHARACTER class must provide some automatic feature to properly zero-terminate the embedded char* string upon return from F77 to C++. This is required in order to 'help' the F77 function to behave almost as if it was a C++ function.
4. The CHARACTER class must provide facilities for passing CHARACTER arrays (i.e. arrays of strings) to F77 functions.

3.5.5.2 single CHARACTER string example

The example below illustrate the most common use of the CHARACTER class, where the C++ code pass simple strings to F77, and receive zero-terminated strings after the call.

first, the F77 subroutine, taking a LOGICAL and a CHARACTER:

```
SUBROUTINE SUB1( FIRST,NAME )
LOGICAL          FIRST
CHARACTER*(*)    NAME
IF( FIRST)THEN
    NAME = 'Elin'
ELSE
    NAME = 'Arnholm'
ENDIF
RETURN
END
```

second, C++ function calling the F77 subroutine (note that the CHARACTER is passed by value, this is an exception from the general rule):

```
#include <fortran.h>
SUBROUTINE SUB1(LOGICAL& FIRST,CHARACTER NAME);
void cppfunc()
{
    const size_t length=10;
    char    name[length];
    strcpy (name,"Bjarne");
```

```

{
    CHARACTER NAME(name,length);
    LOGICAL FIRST = TRUE;
    SUB1( FIRST,NAME );

}
.
.
.
.
.
}

```

Note that the example above will compile, link and run directly under MS Visual C++ and MS FORTRAN Powerstation compilers running Windows 95/NT. The same code will compile unchanged on a UNIX box like the Silicon Graphics (SGI), but will require some stub code to be linked with the application, in order to handle that platform's different way of passing CHARACTER strings. That stub code can be automatically generated, with no manual coding. Please refer to [section 4](#), for further details.

3.5.5.3 Passing single-dimension CHARACTER array parameters

F77 represents a CHARACTER array in basically the same way as a single CHARACTER string, the second array element follows right after the first element etc. The whole array is stored within a contiguous block of memory. The facilities for passing an array of CHARACTER strings can be illustrated through the following example:

first, the F77 subroutine, taking an array of CHARACTER strings:

```

SUBROUTINE SUB1AR( NAME )
  CHARACTER*( *)      NAME( 2 )
  NAME(1) = 'Elin'
  NAME(2) = 'Arnholm'
  RETURN
END

```

second, C++ function calling the F77 subroutine:

```
#include <iostream.h>
#include <fortran.h>
SUBROUTINE SUB1AR(CHARACTER NAME);
void main()
{

    const size_t length=15,arrlen=2;
    char  name[length*arrlen];

    char* firstname = NULL;
    char* lastname = NULL;
    {
        CHARACTER NAME(name,length);

        NAME(0) = "Bjarne";
        NAME(1) = "Stroustrup";

        SUB1AR(NAME);

        firstname = NAME(0);
        lastname  = NAME(1);
    }

    cout << "Name returned : " << firstname << ' ' << lastname
```

```
<< endl;
}
```

3.6 Linking C++ and FORTRAN

This may cause you some practical problems. The solution is typically compiler/platform dependent, but here are some bullet points to keep in mind:

- On any platform
If your F77 code has any BLOCK DATA routines to initialise common blocks, be sure to link these explicitly via object files (to keep them in object libraries is generally not sufficient, because these routines are never called explicitly).
- On a Unix platform
It is usually a good idea to compile the FORTRAN code into object code, and then use the C++ compiler/linker to link the whole program. If you do this, be sure to name all the FORTRAN run-time libraries when linking (tip: on SGI the libraries were `libF77.a`, `libI77.a` and `libISAM.a`). You could do it the other way around, but be prepared to figure out which C++ libraries to include.
- On a PC running Windows 95/NT
If you are using MS VC++ 4.x and Fortran Powerstation: You should probably use multithreading as this seems to eliminate linking problems that occur for singlethreaded code (don't ask me why), and set the Fortran and C/C++ compiler options accordingly. You may also have to struggle a bit with the system libraries (what you need is dependent on the type of application you have). Below is shown some settings for MS VC++ 4.0 and MS Fortran Powerstation 4.0. I used these when linking the examples in [section 6](#) into "console applications":

Suggested Microsoft Developer Studio 4.x settings		
Build->Settings	Fortran tab: Category = "Fortran Libraries"	Use run-time libraries = Multithreaded
	C/C++ tab: Category = "Code Generation"	Use run-time library = Multithreaded
	Link tab: Category = "Input"	Ignore libraries = libcmtd.lib

The notion of portable code should be reserved to those programs that can be moved to another platform/operating system and recompiled without manual code changes. Of course, the program must produce the same results as on the original platform, to claim portability.

With these requirements in mind, how is it possible to circumvent the problems of different linkage and calling conventions, upper and lowercase subroutine names with leading and/or trailing underscores, and more significantly the different ways of passing CHARACTER strings?

The answer lies partly in the fact that all linkage and calling conventions have been hidden inside macro definitions, stored in the central header file, `FORTTRAN.H`. What remains is to deal with the upper/lowercase names, the leading/trailing underscores, and most significantly, passing of CHARACTER strings.

4.1 The idea of stub code generation

One way of solving this, is to create a utility program that can read the prototypes representing F77 subroutines and generate the required "glue" between C++ and F77. Such "glue" is also called "stub code".

We do not want such stub code to interfere with our C++ or FORTRAN application code. Basically, we do not want to see it at all, as it has nothing to do with the real work performed in our application programs. It is therefore important that the prototype definitions representing F77 subroutines are kept in header files separated from other code. When porting our mixed language application (say from Windows NT/95 to UNIX), we will generate new header files containing stub code that suit the target environment. By organising things this way, we can achieve portability without touching neither the C++ nor the FORTRAN application code! Anyone who has experience with maintenance of software targeting several platforms will understand the significance of this.

4.2 Header files containing FORTRAN prototype declarations

When preparing for portable re-use of a set of F77 subroutines, the prototypes must be stored in a separate header file. Below is an example of a user-written header file, here called `"sifttool.h"`. Please note that even if this example presents no parameter names, it is recommended. The prototypes are much more readable that way.

```

#ifndef SIFT00L_H
#define SIFT00L_H
#include <fortran.h>

SUBROUTINE SIFTOL( INTEGER&, INTEGER&, INTEGER&, INTEGER*,
                  INTEGER&, INTEGER*, INTEGER& );

SUBROUTINE
OPEN73( INTEGER&, CHARACTER, CHARACTER, CHARACTER, CHARACTER );
SUBROUTINE CLOS73( INTEGER&, INTEGER& );
SUBROUTINE SIFREH( CHARACTER, CHARACTER, INTEGER&, CHARACTER,

CHARACTER, INTEGER&, INTEGER&, INTEGER&, INTEGER& );
SUBROUTINE
SIFGLS( INTEGER&, CHARACTER, INTEGER&, INTEGER*, INTEGER&, INTEGER& );
SUBROUTINE
SIFSIN( INTEGER*, INTEGER&, CHARACTER, CHARACTER, CHARACTER, CHARACTER,

CHARACTER, CHARACTER, INTEGER&, INTEGER&, INTEGER&, INTEGER& );
SUBROUTINE CSEL73( INTEGER&, INTEGER& );
SUBROUTINE GRES73( CHARACTER, INTEGER&, INTEGER*, INTEGER&,
                  REAL*, INTEGER&, INTEGER& );
SUBROUTINE GREC73( CHARACTER, INTEGER&, REAL*, INTEGER&, INTEGER& );
SUBROUTINE PRES73( CHARACTER, INTEGER&, REAL*, INTEGER&, INTEGER& );
SUBROUTINE
GNRC73( CHARACTER, INTEGER&, INTEGER*, INTEGER&, INTEGER& );
SUBROUTINE
SIFSEC( CHARACTER, INTEGER&, CHARACTER, INTEGER&, REAL*, INTEGER& );
SUBROUTINE
SIFGF1( CHARACTER, INTEGER&, INTEGER&, INTEGER&, INTEGER*, INTEGER& );
SUBROUTINE
SIFHIR( CHARACTER, CHARACTER, CHARACTER, CHARACTER, CHARACTER,

INTEGER&, CHARACTER, INTEGER&, INTEGER&, INTEGER&, INTEGER& );

#endif

```

4.3 Automatic stub code generation based on prototype header files

Header files similar to the one shown in the previous section can be used directly under MS Visual C++ and FORTRAN Powerstation. It will compile and link without any problems.

Many UNIX systems behave differently, however, and will not automatically accept the code as presented. The C++ application code will compile, but the linker will most likely not find the F77 subroutines (because it may expect lowercase subroutine names, possibly with some extra "decoration"). Even if the linker does not detect any problems, the program may crash, because parameters are not passed properly (and this goes undetected because we use extern "C" via the SUBROUTINE macro) .

This is why we need stub code. When calling an F77 subroutine from C++ in a UNIX environment, we will actually call the stub code instead of the FORTRAN code. The stub code then immediately calls the FORTRAN code in the suitable platform dependant way. Typically, all such stub code is declared inline, which eliminates the extra function call overhead. Below is shown generated stub code for the SGI UNIX platform, based on the header file presented in the previous section:

```
#ifndef SIFT00L_H_F77_STUB
#define SIFT00L_H_F77_STUB

#define F77_STUB_REQUIRED
#include <fortran.h>

SUBROUTINE_F77  siftol_(int&,int&,int&,int*,int&,int*,int&);
SUBROUTINE SIFTOL(INTEGER& v1,INTEGER& v2,INTEGER& v3,INTEGER*
v4,
                INTEGER& v5,INTEGER* v6,INTEGER& v7)
{ siftol_(v1,v2,v3,v4,v5,v6,v7); }

SUBROUTINE_F77
open73_(int&,char*,char*,char*,char*,int,int,int,int);
SUBROUTINE OPEN73(INTEGER& v1,CHARACTER v2,CHARACTER v3,
                CHARACTER v4,CHARACTER v5)
```



```

    {
open73_(v1,v2.rep,v3.rep,v4.rep,v5.rep,v2.len,v3.len,v4.len,v5.len);
    }

SUBROUTINE_F77 clos73_(int&,int&);
SUBROUTINE CLOS73(INTEGER& v1,INTEGER& v2)
    { clos73_(v1,v2); }

SUBROUTINE_F77
sifreh_(char*,char*,int&,char*,char*,int&,int&,int&,int&,
        int,int,int,int);
SUBROUTINE SIFREH(CCHARACTER v1,CHARACTER v2,INTEGER& v3,
        CHARACTER v4,CHARACTER v5,INTEGER&
v6,INTEGER& v7,
        INTEGER& v8,INTEGER& v9)
    { sifreh_(v1.rep,v2.rep,v3,v4.rep,v5.rep,v6,v7,v8,v9,
        v1.len,v2.len,v4.len,v5.len); }

SUBROUTINE_F77 sifgls_(int&,char*,int&,int*,int&,int&,int);
SUBROUTINE SIFGLS(INTEGER& v1,CHARACTER v2,INTEGER& v3,
        INTEGER* v4,INTEGER& v5,INTEGER& v6)
    { sifgls_(v1,v2.rep,v3,v4,v5,v6,v2.len); }

SUBROUTINE_F77
sifsin_(int*,int&,char*,char*,char*,char*,char*,char*,
        int&,int&,int&,int&,int,int,int,int,int,int);
SUBROUTINE SIFSIN(INTEGER* v1,INTEGER& v2,CHARACTER v3,
        CHARACTER v4,CHARACTER v5,CHARACTER v6,
        CHARACTER v7,CHARACTER v8,INTEGER& v9,
        INTEGER& v10,INTEGER& v11,INTEGER& v12)
    {
sifsin_(v1,v2,v3.rep,v4.rep,v5.rep,v6.rep,v7.rep,v8.rep,v9,
        v10,v11,v12,v3.len,v4.len,v5.len,v6.len,v7.len,v8.len); }

```

```

SUBROUTINE_F77  csel73_(int&,int&);
SUBROUTINE CSEL73(INTEGER& v1,INTEGER& v2)
    { csel73_(v1,v2); }

SUBROUTINE_F77
gres73_(char*,int&,int*,int&,float*,int&,int&,int);
SUBROUTINE GRES73(CHARACTER v1,INTEGER& v2,INTEGER* v3,
    INTEGER& v4,REAL* v5,INTEGER& v6,INTEGER& v7)
    { gres73_(v1.rep,v2,v3,v4,v5,v6,v7,v1.len); }

SUBROUTINE_F77  grec73_(char*,int&,float*,int&,int&,int);
SUBROUTINE GREC73(CHARACTER v1,INTEGER& v2,REAL* v3,INTEGER&
v4,INTEGER& v5)
    { grec73_(v1.rep,v2,v3,v4,v5,v1.len); }

SUBROUTINE_F77  pres73_(char*,int&,float*,int&,int&,int);
SUBROUTINE PRES73(CHARACTER v1,INTEGER& v2,REAL* v3,INTEGER&
v4,INTEGER& v5)
    { pres73_(v1.rep,v2,v3,v4,v5,v1.len); }

SUBROUTINE_F77  gnrc73_(char*,int&,int*,int&,int&,int);
SUBROUTINE GNRC73(CHARACTER v1,INTEGER& v2,INTEGER* v3,
    INTEGER& v4,INTEGER& v5)
    { gnrc73_(v1.rep,v2,v3,v4,v5,v1.len); }

SUBROUTINE_F77
sifsec_(char*,int&,char*,int&,float*,int&,int,int);
SUBROUTINE SIFSEC(CHARACTER v1,INTEGER& v2,CHARACTER v3,
    INTEGER& v4,REAL* v5,INTEGER& v6)
    { sifsec_(v1.rep,v2,v3.rep,v4,v5,v6,v1.len,v3.len); }

SUBROUTINE_F77  sifgf1_(char*,int&,int&,int&,int*,int&,int);
SUBROUTINE SIFGF1(CHARACTER v1,INTEGER& v2,INTEGER& v3,
    INTEGER& v4,INTEGER* v5,INTEGER& v6)
    { sifgf1_(v1.rep,v2,v3,v4,v5,v6,v1.len); }

```

```

SUBROUTINE_F77
sifhir_(char*,char*,char*,char*,char*,int&,char*,int&,

int&,int&,int&,int,int,int,int,int,int);
SUBROUTINE SIFHIR(CHARACTER v1,CHARACTER v2,CHARACTER v3,
                  CHARACTER v4,CHARACTER v5,INTEGER& v6,
                  CHARACTER v7,INTEGER& v8,INTEGER& v9,INTEGER&
v10,
                  INTEGER& v11)
{ sifhir_(v1.rep,v2.rep,v3.rep,v4.rep,v5.rep,v6,v7.rep,

v8,v9,v10,v11,v1.len,v2.len,v3.len,v4.len,v5.len,v7.len); }

#endif

```

4.3.1 The HCOMP utility program

The process of porting mixed language applications can be summarised as follows (the description assumes that the base platform is Windows NT/95 and that the application is ported to a UNIX platform such as SGI).

1. Develop and test the application under Windows

1. Write header files for each of your FORTRAN libraries
2. Write your C++ application code
3. Build and test the application

2. Port the application

1. Create stub code for each target platform
1. Run HCOMP on each of the FORTRAN header files
2. Copy the header files generated by HCOMP to the target platform
2. Copy your C++ and FORTRAN application code to the target platform
3. Compile the C++ code on the target platform (make sure header files generated by HCOMP are used instead of the original ones).
4. Compile the FORTRAN code

5. Link and test the application

If all goes well, you should now have an application which runs equally well on both platforms. The source code of the program HCOMP is found in the following files:

- HCOMP.CPP - the main program
- FTYPE.H - declaration of 'ftype', a class holding information about each call parameter
- FTYPE.CPP - implementation of ftype member functions
- HCUTIL.H - header file for string manipulation functions
- HCUTIL.CPP - implementation of some useful string manipulation functions

HCOMP is currently not a portable program in itself. It must be compiled and executed under Windows 95/NT.

The `fortran.h` file is included from the header files declaring C++ prototypes for F77 subroutines and functions. The other include files are automatically included from here.

```
#ifndef FORTRAN_FROM_CPLUSPLUS
#define FORTRAN_FROM_CPLUSPLUS
```

```
typedef int    INTEGER;
typedef float  REAL;
typedef double DOUBLE_PRECISION;
typedef int    LOGICAL;
```

```
#include <f77char.h>
#include <f77cmplx.h>
#include <f77matrx.h>
```

```
#define FALSE 0
#define TRUE  1
```

```
#ifdef F77_STUB_REQUIRED
```

```

#define SUBROUTINE                inline void
#define INTEGER_FUNCTION          inline INTEGER
#define REAL_FUNCTION             inline REAL
#define LOGICAL_FUNCTION         inline LOGICAL
#define DOUBLE_PRECISION_FUNCTION inline
DOUBLE_PRECISION

#define SUBROUTINE_F77            extern "C" void
#define INTEGER_FUNCTION_F77      extern "C" int
#define REAL_FUNCTION_F77         extern "C" float
#define LOGICAL_FUNCTION_F77      extern "C" int
#define DOUBLE_PRECISION_FUNCTION_F77 extern "C" double
#else

```

```

#define SUBROUTINE                extern "C" void
__stdcall
#define INTEGER_FUNCTION          extern "C" INTEGER
__stdcall
#define REAL_FUNCTION             extern "C" REAL
__stdcall
#define LOGICAL_FUNCTION         extern "C" LOGICAL
__stdcall
#define DOUBLE_PRECISION_FUNCTION extern "C"
DOUBLE_PRECISION \

__stdcall
#endif
#endif

```

5.1 The COMPLEX class

The COMPLEX class is available via the `fortran.h` include file.

```
#ifdef real

    #undef real
    #pragma message(__FILE__" : warning: 'real' macro definition
cancelled")
#endif

template<class T>
class COMPLEX {
public:
    COMPLEX();
    COMPLEX(const COMPLEX<T>& );
    COMPLEX(const T& re,const T& im);
    COMPLEX<T>& operator=(const COMPLEX<T>& );
    ~COMPLEX();
    const T& real();
    const T& imag();
private:
    T m_re;
    T m_im;
};

template<class T>
inline COMPLEX<T>::COMPLEX()
:m_re(T()),m_im(T())
{}

template<class T>
inline COMPLEX<T>::COMPLEX(const COMPLEX<T>& copy)
:m_re(copy.m_re),m_im(copy.m_im)
{}

template<class T>
```

```

inline COMPLEX<T>::COMPLEX(const T& re,const T& im)
:m_re(re),m_im(im)
{}

template<class T>
inline COMPLEX<T>& COMPLEX<T>::operator=(const COMPLEX<T>&
copy)
{
    m_re = copy.m_re;
    m_im = copy.m_im;
    return *this;
}

template<class T>
inline COMPLEX<T>::~~COMPLEX( )
{}

template<class T>
inline const T& COMPLEX<T>::real( )
{
    return m_re;
}

template<class T>
inline const T& COMPLEX<T>::imag( )
{
    return m_im;
}

```

5.2 The CHARACTER class

The CHARACTER class is available via the `fortran.h` include file.
`#include <string.h>`

```

class CHARACTER {
public:
    CHARACTER(char* cstring);
    CHARACTER(char* cstring, const size_t lstr);
    ~CHARACTER();
    CHARACTER operator()(size_t index);
    void pad(size_t first,size_t howmany=1);
    void operator=(char* str);
    operator char*();
public:
    char*   rep; // Actual string
    size_t  len; // String length
};

inline CHARACTER::CHARACTER(char* cstring)
: rep(cstring), len(strlen(cstring))
{};

inline CHARACTER::CHARACTER(char* cstring, const size_t lstr)
: rep(cstring), len(lstr)
{

    size_t slen  = strlen(rep);
    size_t actual = (slen < len)? slen : len;
    for(size_t i=actual;i<len;i++) rep[i]=' ';
}

inline CHARACTER::~CHARACTER() {
    if(rep[len] == '\\0') return;

    for(int i=len-1;i>=0;i--) {
        if(rep[i] == '\\0') break;

        if(rep[i] != ' ') {
            rep[i+1] = '\\0';
            break;
        }
    }
}

```



```

    }
}

inline CHARACTER CHARACTER::operator()(size_t index)
{

    size_t pos = index*len;
    CHARACTER element(rep+pos,len);
    return element;
}

inline void CHARACTER::pad(size_t first,size_t howmany)
{

    size_t pos=0,i=0,stop=first+howmany-1;
    for(size_t index=first; index<=stop; index++) {
        pos = index*len;
        size_t slen = strlen(rep+pos);
        size_t actual = (slen < len)? slen : len;
        for(i=pos+actual;i<pos+len;i++) rep[i]=' ';
    }
}

inline void CHARACTER::operator=(char* str)
{
    strncpy(rep,str,len);
    rep[len-1] = '\\0';
    size_t slen = strlen(rep);
    size_t actual = (slen < len)? slen : len;
    for(size_t i=actual;i<len;i++) rep[i]=' ';
}

inline CHARACTER::operator char*()
{

```

```

        return rep;
    }

```

5.3 The FMATRIX class

The FMATRIX class is available via the `fortran.h` include file.

```
#include <assert.h>
```

```

template <class T>
class FMATRIX {
public:
    FMATRIX(size_t dim1, size_t dim2);
    FMATRIX(T* cpparr, size_t dim1, size_t dim2);
    operator T*();
    T& operator()(size_t index1, size_t index2);
    ~FMATRIX();
public:
    const size_t ndim;
    size_t dim[7];
    T*  cpprep;
    T*  f77rep;
};

```

```

template <class T>
FMATRIX<T>::FMATRIX(size_t dim1, size_t dim2)
: cpprep(NULL), f77rep(new T[dim1*dim2]), ndim(2)
{
    dim[0]=dim1;
    dim[1]=dim2;
    dim[2]=0;
    dim[3]=0;
    dim[4]=0;
    dim[5]=0;
    dim[6]=0;
}

```

```

}

template <class T>
FMATRIX<T>::FMATRIX(T* cpparr, size_t dim1, size_t dim2)
: cppprep(cpparr),f77rep(new T[dim1*dim2]),ndim(2)
{
    dim[0]=dim1;
    dim[1]=dim2;
    dim[2]=0;
    dim[3]=0;
    dim[4]=0;
    dim[5]=0;
    dim[6]=0;

    size_t index_cpp=0;
    size_t index_f77;
    for(size_t i=0;i<dim[0];i++) {
        for(size_t j=0;j<dim[1];j++) {
            index_f77 = j*dim[0] + i;
            f77rep[index_f77] = cppprep[index_cpp++];
        }
    }
}

template <class T>
FMATRIX<T>::operator T*( )
{
    return f77rep;
}

template <class T>
T& FMATRIX<T>::operator()(size_t index1, size_t index2)
{
    assert(ndim==2);

```

```

        size_t index_f77 = index2*dim[0] + index1;

        return *(f77rep+index_f77);
    }

template <class T>
FMATRIX<T>::~FMATRIX( )
{
    if(cpprep) {
        assert(ndim==2);

        size_t index_cpp;
        size_t index_f77=0;
        for(size_t j=0;j<dim[1];j++) {
            for(size_t i=0;i<dim[0];i++) {
                index_cpp = i*dim[1] + j;
                cpprep[index_cpp] = f77rep[index_f77++];
            }
        }
    }

    delete[] f77rep;
}

```

6.1 CHARACTER example

The [sections 3.5.5, "Passing single-value CHARACTER parameters"](#) and [3.5.5.3, "Passing single-dimension CHARACTER array parameters"](#) demonstrated in detail the most common ways the CHARACTER class is used. We therefore present a slightly more complex (and uncommon) use of the CHARACTER class.

6.1.1 How to call a function returning a CHARACTER

The following example is similar to the COMPLEX FUNCTION example in [section 6.2.1](#), in that calls to such functions are not as portable as calls to FUNCTIONS returning INTEGER, REAL, LOGICAL or DOUBLE PRECISION. The only true portable way of calling COMPLEX or CHARACTER functions is to make F77 wrapper SUBROUTINES and call these from C++ instead. To write such SUBROUTINES is trivial. The following example illustrates an alternative technique, which is less trivial and also machine dependent. The advantage of this alternative approach is that the C++ application code can use a more natural syntax.

Consider the following standard F77 CHARACTER*80 FUNCTION:

```
CHARACTER*80 FUNCTION SECTIM( ISECS )
INTEGER                                ISECS

CHARACTER*80 STRING
INTEGER      IH,IM,IS,I

  IH = ISECS/3600
  IM = ( ISECS - IH*3600 )/60
  IS = ISECS - IH*3600 - IM*60
  WRITE( STRING,100 ) IH,IM,IS
100  FORMAT( I2, ': ', I2, ': ', I2 )
  DO 1000 I=1,8
      IF( STRING( I:I ) .EQ. ' ' ) STRING( I:I )='0'
1000  CONTINUE

  SECTIM = STRING
  RETURN
END
```

The function generates a string containing a time value in the format HH:MM:SS. The problem is to call this function from C++. The following header file (called `sectim.h`) illustrates one possible solution:

```
#ifndef SECTIM_H
#define SECTIM_H
```

```
#include <fortran.h>
```

```
SUBROUTINE SECTIM(Character function_value, INTEGER& isecs);
```

```
inline char* sectim(INTEGER& isecs)
{
```

```
    const size_t string_len=81;
    static char  static_string[string_len];
```

```
    CHARACTER function_value(static_string,string_len);
    SECTIM(function_value, isecs);
```

```
    return static_string;
}
```

```
#endif
```

We can now call the function as illustrated in this example application:

```
#include "iostream.h"
#include "sectim.h"
```

```
int main( )
{
    INTEGER isecs = 60*60*5 + 60*15;
```

```

const size_t tlen=81;
char time[tlen];
{

    CHARACTER TIME(time,tlen);
    TIME = sectim(isecs);
}
cout << isecs << " seconds from midnight at "
    << time << " precisely" << endl;
return 0;
}

```

The output from the program becomes:

18900 seconds from midnight at 05:15:00 precisely

6.2 COMPLEX example

This example illustrates

- How to pass a simple COMPLEX value as a parameter
- How to pass a one-dimensional COMPLEX array as parameter
- How to return a COMPLEX as a function value

All of this is combined in the following section:

6.2.1 How to call a function returning a COMPLEX

In [section 3.5.4](#), the problems relating to functions returning COMPLEX values are discussed. This example illustrates how this problem can be overcome, by relaxing the demands on single-source portability. The platform dependent code can be isolated to header files. Conditional compilation techniques using `#ifdef` for parts of the header file code is probably an acceptable cost in this case.

This example also introduces some features of the new C++ Standard Library. The C++ Standard Library contains among other things the Standard Template Library (STL), the standard `string` class, as well as the standard `complex` class. The example will illustrate how the FORTRAN-compatible COMPLEX class can live side by side with the `complex` class within the same application. Consider the following standard F77 COMPLEX FUNCTION:

```

      COMPLEX FUNCTION ZSUM(NUM,ZARR)
      INTEGER          NUM
      COMPLEX          ZARR(NUM)
      INTEGER I
      COMPLEX SUM
      SUM = 0.D0
      DO 100 I = 1, NUM
        SUM = SUM + ZARR(I)
100    CONTINUE
      ZSUM = SUM
      RETURN
      END

```

The function ZSUM calculates the sum of all elements in an array of complex values.

How can we call this function from C++, without introducing platform dependent C++ code in our application calling the function ZSUM? The following header file (called zsum.h) illustrates one possible solution:

```

#ifndef ZSUM_H
#define ZSUM_H
#include <fortran.h>

```

```

SUBROUTINE ZSUM(COMPLEX<REAL>& function_value,
               const INTEGER& NUM, COMPLEX<REAL>* ZARR);

inline COMPLEX<REAL> zsum(const INTEGER& NUM, COMPLEX<REAL>*
ZARR)
{
    COMPLEX<REAL> SUM;
    ZSUM(SUM,NUM,ZARR);
    return SUM;
}

```



```
}
```

```
#endif
```

Now we have a portable interface towards the COMPLEX FUNCTION (the implementation of the interface is machine dependent, however). The following example application calls the function. The example application is also using the C++ Standard Library class `complex`, to illustrate how the two classes can live side-by-side (one could argue that the COMPLEX class is not needed, since the `complex` class probably stores the same values, but then we would be assuming something about the internal layout of one of the C++ Standard Library classes. We would also silently be assuming that the `complex` class had no virtual member functions, now or in the future. Making such assumptions would be constructing a time-bomb):

```
// standard C++ headers
#include <complex>
#include <iostream>

// The header file for our COMPLEX FUNCTION
#include "zsum.h"

int main()
{

    const INTEGER NZARR=3;
    COMPLEX<REAL> ZARR[NZARR];

    for (int i=0; i<NZARR; i++) {
```

```

    REAL rval = REAL(i+1);
    REAL ival = -rval*rval;
    ZARR[i] = COMPLEX<REAL>(rval,ival);

    COMPLEX<REAL> value(rval,ival);
    cout << "ZARR["<<i<<"] = " << value << endl;
}

COMPLEX<REAL> SUM = zsum(NZARR, ZARR);

complex<float> sum(SUM.real(),SUM.imag());
cout << "-----" << endl
     << "sum      = " << sum << endl;

return 0;
}

```

The program produces the following output (notice how easy complex value I/O becomes with iostream and the new standard complex class):

```

ZARR[0] = (1,-1)
ZARR[1] = (2,-4)
ZARR[2] = (3,-9)
-----
sum      = (6,-14)

```

6.3 FMATRIX, a 2-dimensional array example

The following example illustrates

- How to pass and receive 2-dimensional REAL array as parameter

The following example shows how to call an F77 routine that does matrix multiplication on two 2-dimensional arrays (A and B) and return the result in a third 2-dimensional array (AB).

The actual problem solved is $AB = A*B$:

```

=====
B: | 1 | 2 |
   | 3 | 4 |
   | 5 | 6 |
=====
A: | 1 | 2 | 3 | 22 | 28 | ( AB, the result of
multiplying A with B
   | 4 | 5 | 6 | 49 | 64 | is shown in bold)
=====
```

```
#include <fortran.h>
#include <iostream.h>
```

```
#define loc2(a) &a[0][0]
```

```

SUBROUTINE MTXMUL(REAL* A, REAL* B,REAL* AB,
                  INTEGER& M,INTEGER& K,INTEGER& N);

int main()
{
    const size_t m=2,k=3,n=2;
    float        a[m][k],b[k][n],ab[m][n];

    a[0][0] = 1;   a[0][1] = 2;   a[0][2] = 3;
    a[1][0] = 4;   a[1][1] = 5;   a[1][2] = 6;

    b[0][0] = 1;   b[0][1] = 2;
    b[1][0] = 3;   b[1][1] = 4;
    b[2][0] = 5;   b[2][1] = 6;

    {
        FMATRIX<REAL>
        A(loc2(a),m,k),B(loc2(b),k,n),AB(loc2(ab),m,n);
        INTEGER        M(m),K(k),N(n);
        MTXMUL(A,B,AB,M,K,N);

    }

    cout << " ab[0][0] = " << ab[0][0] << " ab[0][1] = " <<
    ab[0][1] << endl;
    cout << " ab[1][0] = " << ab[1][0] << " ab[1][1] = " <<
    ab[1][1] << endl;

    return 0;
}

```

The program produces the following (correct!) output:

```
ab[0][0] = 22 ab[0][1] = 28
ab[1][0] = 49 ab[1][1] = 64
```

The F77 subroutine MTXMUL, called from the C++ program is shown below:

```
      SUBROUTINE MTXMUL (A,      B,      AB,M,K,N)
      INTEGER                      M,K,N
      REAL                        A(M,K),B(K,N),AB(M,N)

      DO 120 I=1,M
      DO 130 J=1,N
          S = 0.0
          DO 140 IJ=1,K
              S = S + A(I,IJ)*B(IJ,J)
140      CONTINUE
          AB(I,J) = S
130 CONTINUE
120 CONTINUE

9999 CONTINUE
      RETURN

      END
```

6.4 Mixed language file I/O example

This example is an extremely simple, yet complete program. It illustrates the problem of performing simultaneous file I/O to the same file from both languages, which is not always easy (or even possible). The example builds on the idea of doing the actual I/O in one language only. In this case F77 is used for the basic I/O. A class 'fostream' implements an fortran I/O-based iostream-like object for use in C++.

First, the main program:

main.cpp

```
#include "fostream.h"
#include "subdmo.h"
int main()
{
    int iu = 10;
    fostream fout(iu,"F77IO.txt");

    fout << "C++:" << "first string" << '\n';

    SUBDMO(iu);

    fout << "C++:" << "second string" << '\n';

    return 0;
}
```

subdmo.h

```
#ifndef SUBDMO_H
#define SUBDMO_H

#include <fortran.h>
SUBROUTINE SUBDMO(INTEGER& IU);

#endif
```

fostream.h

```
#ifndef FOSTREAM_H
#define FOSTREAM_H

#include <stdlib.h>
class fostream {
public:
    fostream(int funit=6, char* filename=NULL);
    fostream& operator <<(const char ch);
    fostream& operator <<(const char* txt);
    ~fostream();
};

#endif
```

subdmo.for

```

        SUBROUTINE SUBDM0(IUNIT)
        INTEGER          IUNIT
        WRITE(IUNIT,10) 'Text written from FORTRAN on unit:
',iunit
10      FORMAT(1x,A35,I2)
        END

```

fostream.cpp

```

#include "fostream.h"

#include <fortran.h>
SUBROUTINE F770PN(const INTEGER& IU,CHARACTER NAME);
SUBROUTINE F770UT(CHARACTER STRING);
SUBROUTINE F77CLS();

fostream::fostream(int funit, char* filename)
{
    if(filename)
        F770PN(funit,CHARACTER(filename));
    else
        F770PN(funit,CHARACTER("stdout.txt"));
}

fostream::~fostream()
{
    F77CLS();
}

fostream& fostream::operator <<(const char ch)

```



```

{
    char str[2] = { ch, '\0' };
    F77OUT(CHARACTER(str));
    return *this;
}

fostream& fostream::operator <<(const char* txt)
{
    F77OUT(CHARACTER((char*)txt));
    return *this;
}

```

f77out.for

```

SUBROUTINE F77OUT(STRING)
CHARACTER*(*)      STRING
INTEGER L1,L2
CHARACTER*512      LINE
SAVE LINE
DATA LINE /' '/
INTEGER            IUNIT
COMMON /FILEIO/ IUNIT
IF(LEN(STRING).EQ.1 .AND. ICHAR(STRING(1:1)).EQ. 10)THEN
    CALL SLEN(LINE,L1)
    WRITE(IUNIT,10) LINE(1:L1)
10    FORMAT(1X,A)
    LINE = ' '
ELSE
    CALL SLEN(LINE,L1)
    CALL SLEN(STRING,L2)
    LINE = LINE(1:L1)//STRING(1:L2)
ENDIF

```

END

SUBROUTINE F77OPN(IU,NAME)

INTEGER IU

CHARACTER*(*) NAME

INTEGER IUNIT

COMMON /FILEIO/ IUNIT

CALL F77CLS()

IF(IU.NE.6)OPEN(UNIT=IU,FILE=NAME,STATUS='UNKNOWN')

IUNIT = IU

END

SUBROUTINE F77CLS()

INTEGER IUNIT

COMMON /FILEIO/ IUNIT

IF(IUNIT.GT.0)THEN

CLOSE(IUNIT)

ENDIF

END

SUBROUTINE SLEN(STRING,LS)

CHARACTER*(*) STRING

INTEGER LS

DO 1000 I=LEN(STRING),1,-1

IF(STRING(I:I).NE. ' ')THEN

LS = I

RETURN

ENDIF

1000 CONTINUE

LS = 0

RETURN

END