

MIXED LANGUAGE PROGRAMMING

Burkhard D. Burow^{a,b}

*Universität Hamburg, II. Institut für Experimentalphysik,
Luruper Chaussee 149, 22761 Hamburg, Germany*

Computing in the next millennium will be using software from this millennium. Programming languages evolve and new ones continue to be created. The use of legacy code demonstrates why some present and future applications may span programming languages. Even a completely new application may mix programming languages, if it allows its components to be more conveniently expressed. Given the need, mixed language programming should be easy and robust. By resolving a variety of difficulties, the well established cfortran.h package provides the desired convenient interface across the C and Fortran programming languages.

This presentation examines mixed language programming. It aims to help programmers of all languages benefit from the possibilities offered by mixed language programming and to help them create software which in turn may enjoy a long and useful life, perhaps at times in a mixed language program. By encouraging and facilitating code reuse and the use of well-suited programming languages, one may help eliminate the qualifier in the maxim:

“Scientists stand on the shoulders of their predecessors,
except for computer scientists, who stand on the toes.”

1 Introduction

1.1 Applications from Components

A computer application consists of components, each supplying part of the action performed by the application. The action provided by any given component may be required by several applications. Therefore, applications reuse components. The expensive alternative to reuse is to recreate components.

1.2 The Need for Mixed Language Programming

A component is coded in a programming language. An existing component is thus coded in one of the various programming languages. Depending on the action performed by the component, it may be best expressed in a particular language. For a component coded in one of the general purpose programming languages, the choice of programming language is strongly determined by the programmer's preferences. The components used by an application may thus span several languages. The creation of such a mixed language application is called Mixed Language Programming (MLP).

1.3 Mixed Language Programming is NOT Translation

For some components and/or languages, it may be possible to translate the component into another language. Translation is sensible when further development and maintenance of the component will take place in the new language and is abandoned in the old language. Such a translation is not an MLP issue.

a. Presented in the parallel session 'E.1 Languages' at Computing in High Energy Physics (CHEP95), Rio de Janeiro, Brazil, 18-22 September 1995.

b. Also supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the BMBF under Contract Nr.056HH29I.

Translation is generally not a method to perform MLP, since MLP is concerned with the use of a component and thus with only its interface or ‘surface’. There is no need to translate the ‘body’ of the component. Additional practical difficulties include:

- The source code of the component may not be available. For example, many commercial components are only available as libraries of compiled objects.
- Due to the different abilities of the languages involved, the translation may be practically impossible. For example, the translation of a component written in C into Fortran is practically impossible, since C has many features not present in Fortran.
- Even where possible, translation is difficult, even with tools.
- Translation introduces opportunities for bugs.

1.4 The Anatomy of a Component

The anatomy of a typical component is sketched in Figure 1a). Within a programming language, the use of components is well-supported: the name is obvious; passing input and output data is easy and there is a consistent environment for any component side-effects.

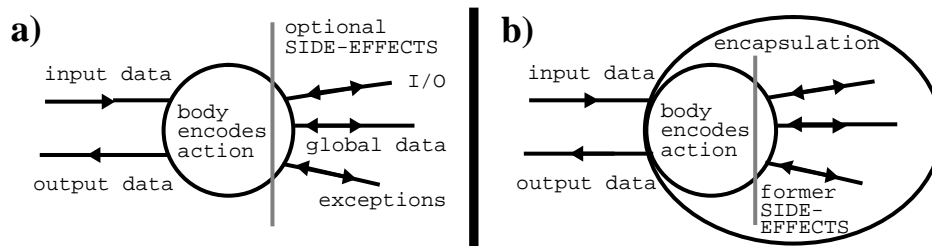


Figure 1 a) The anatomy of a typical component. b) Effectively eliminating side-effects by encapsulation.

1.5 The Trouble with Mixed Language Programming

Crossing languages, the use of components is generally NOT well-supported: a component’s name may not be obvious; passing input and output data may be complicated and there is a potentially troubled environment for any component side-effects.

1.6 Avoiding Trouble with Side-Effects

Following good programming practice, side-effects are best avoided since they are generally trouble-some, not just for MLP. For example, instead of printing an error message, a component should return an error code among the output data. As sketched in Figure 1b), encapsulation may allow an existing component to be effectively freed of side-effects.

2 Three Types of Mixed Language Programming

2.1 Coarse-Grained Mixed Language Programming

For coarse-grained MLP, the application consists of multiple executables. Each executable consists of components written in a single language. The basic notion is that each executable works on an intermediate file of data. Once the work has completed, the next executable may work on the file of data. The scheme can enjoy two refinements. Scripts can automate running the executables. The intermediate file(s) can often be replaced by a flow of data through a pipe or pipes connecting the executables.

Coarse-grained MLP has several advantages and is an obvious method if one of the executables already exists. It has none of the MLP problems, other than passing data between executables written in different languages. Not just for MLP, but in general, the separate executables should simplify the development and maintenance of the application.

Coarse-grained MLP is unfortunately of limited applicability. It is unsuitable for applications which thickly interleave the use of components across different languages.

2.2 CORBA

The Common Object Request Broker Architecture (CORBA) is a powerful, generic environment for combining components. For component use, the language of the component is irrelevant once the language is bound to the Interface Definition Language (IDL). The IDL solves the MLP problems for component names and for passing input and output data. Experience with CORBA for MLP is presented at CHEP95 by Quarrie.

2.3 Mixed Language Executables

A Mixed Languages Executable (MLX) is perhaps the most common form of MLP. For example, C and Fortran routines are combined to create CERN's popular PAW¹ program.

Unfortunately few programming languages provide standard support for MLX. An exception is C++, which supports the use of C routines. In future, Fortran 2000 may provide standard 'interoperability with C'.

Despite the dearth of standard support, MLX programming is possible. The remainder of this presentation examines the difficulties of MLX and their solution.

3 Mixed Language Executables

3.1 Direct MLX ?

In the most direct form of MLX, the machine dependent recipe to call a foreign routine is contained in the application code for each foreign routine and for each machine. As demonstrated by the code fragment in Figure 2a), direct MLX can be tedious, cluttered and error-prone. Moreover, the troubles geometrically worsen with the complexity of the routines' interfaces and with the number of machines and calls to foreign routines. Nevertheless, direct MLX may be tolerable for applications running on a single type of machine, using routines with simple interfaces or when nicer methods, see below, are unavailable.

3.2 Wrappers for MLX

As illustrated in Figure 2b), all MLX problems disappear for the application programmer once a wrapper provides a native interface to the foreign routine. A native interface implies that the use of the foreign routine via the wrapper is machine independent, with a simple name and easy passing of input and output arguments. For any foreign routine, a wrapper has only to be created once for each machine type.

3.3 Direct Wrappers ?

Wrappers may of course be directly created by the programmer, but this shares many of the troubles of direct MLX. As illustrated in Figure 2c), considerable tedious effort has to be exerted to create a wrapper for each routine and machine.

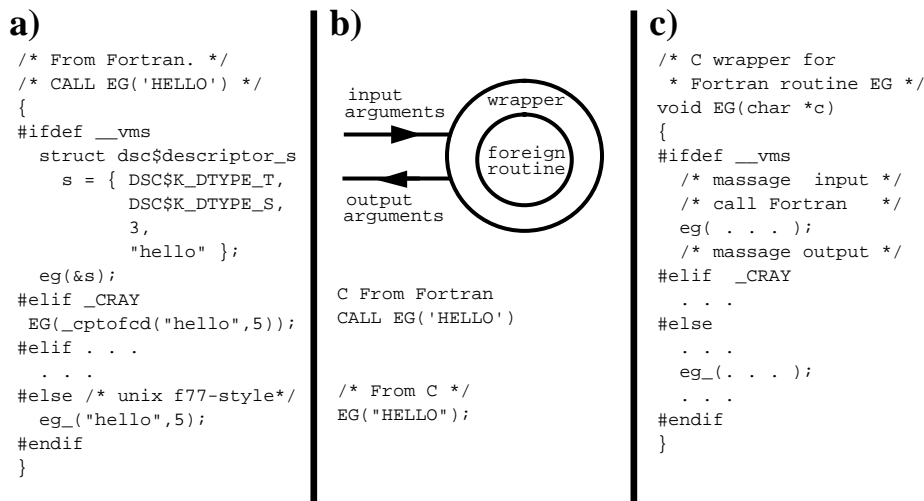


Figure 2 a) An example code fragment of C calling Fortran using direct MLX.
b) An illustration of using a wrapper for MLX.
c) An example of a direct wrapper to allow C to easily call a Fortran routine.

3.4 Machine Independent Wrapper Specification

A wrapper may be easily created once a machine's MLX recipe has been encoded in some form. As illustrated in Figure 3a), the creation of the wrapper thus only requires the encoding and a machine independent specification of the wrapper.

3.5 *cfortran.h*

MLX recipes have been encoded into various 'wrapper schemes'. For combining C and Fortran², *cfortran.h*³ is the most complete scheme. It is available via anonymous ftp at zebra.desy.de. *cfortran.h* supports:

- C calling Fortran. Fortran calling C.
- Names of routines and common blocks.
- Input and output arguments. Function return values.
- Most data types and is extensible with simple user defined types.
- VMS, OSF, Ultrix, AIX, IRIX, CRAY, SunOS, Apollo, HP-UX, Convex, . . .

3.6 *hbook.h*

The wrapper specifications using *cfortran.h* are typically in a C header file. A popular example is *hbook.h*, fragments of which are shown in Figure 3b). As demonstrated by the example C program in Figure 3c), *hbook.h* allows C to easily access CERN's HBOOK library of Fortran routines. C access to other CERNLIB Fortran libraries is provided by *adamo.h*, *geant321.h*, *minuit.h*, etc. The *f2h* utility in CERNLIB can 'automatically' create the *.h header files from Fortran source code.

a)

```
#include "cfortran.h"    /* encoding */

/* machine independent wrapper */
#define EG(C) \
    CCALLSFSUB1(EG,eg, STRING, C)

main()
{
    EG("HELLO"); /* call Fortran routine */
}
```

b)

```
/* hbook.h of cernlib@cern.ch */
. . .
#define HBARX(A1) \
    CCALLSFSUB1(HBARX,hbarx,INT,A1)
. . .
#define HISTDO() \
    CCALLSFSUB0(HISTDO,histdo)
. . .
```

c)

```
/* An example of HBOOK use from C. */
#include "cfortran.h"
#include "hbook.h"

#define NWPAWC 50000
typedef float PAWC_DEF[NWPAWC];
#define PAWC COMMON_BLOCK(PAWC,pawc)
COMMON_BLOCK_DEF(PAWC_DEF,PAWC);
PAWC_DEF PAWC;

main() {
    int i, id=10;

    HLIMIT(NWPAWC);
    HBOOK1(id,"1-DIM",100,1,101,0);

    for ( i=1; i<=100; i++)
        HFILL(id,i+.5, 0, 10*(i%25) );

    HISTDO();
}
```

Figure 3 a) An example C program creating and using a machine independent wrapper specification.

b) Fragments from the C header file hbook.h, providing C wrappers to the Fortran HBOOK routines.

c) An example C program using CERN's HBOOK library of Fortran routines.

3.7 HERMES and cfortran.h

The HERMES collaboration at HERA is bilingual, though most programmers write either C or Fortran. As of one of the largest and most successful HEP users of cfortran.h, HERMES enjoys comfortable and productive use of both C and Fortran throughout their software development including the use of outside libraries and the creation of HERMES libraries and applications⁴.

4 Conclusion

Facilitating and encouraging MLP promotes code reuse and the use of languages suitable for the application and the programmer.

C and Fortran are excellent examples of how MLP has replaced previous rivalry by cooperation between languages.

cfortran.h allows HERMES, PAW and many others to enjoy C and Fortran.

References

1. J. Bunn, "A New Query Processor for PAW", at CHEP95.
2. A. Nathaniel, "Interfacing C and Fortran", CERN COMPUTER NEWSLETTER No. 217 (July - September 1994) p.9. This is an excellent introduction to the MLX recipes required to combine C and Fortran.
3. Paul F. Dubois, Lee Busby, "Portable, Powerful Fortran Programs", *Computers in Physics*, Vol. 7, No. 1, Jan./Feb. 1993. This includes an introduction to cfortran.h.
4. W. Wander, "DAD - Distributed ADAMO Database system at Hermes", at CHEP95.
K. Ackerstaff, "A Tcl/Tk Database Interface to ADAMO and DAD", at CHEP95.