CALLING FORTRAN SUBROUTINES FROM FORTRAN, C, AND C++

FRITZ KEINERT
DEPARTMENT OF MATHEMATICS
IOWA STATE UNIVERSITY
AMES, IA 50011
KEINERT@IASTATE.EDU

Contents

1. Introduction

Virtually all scientific computation makes use of standard ("canned") subroutines at some point. In this report, I have written up some general guidelines and a number of examples for linking Fortran subroutines to main programs written in Fortran, C, or C++. There used to be another section on calling Fortran subroutines from Matlab. I have decided to put that into a separate document because this is considerably harder and probably of less interest to most people. All examples use the CMLIB and LAPACK libraries.

The report is intended as a resource for research activities at Iowa State, and as a supplementary reference for numerical analysis courses.

The case studies were developed and tested on the three different hardware platforms that currently make up Project Vincent. Project Vincent is a large network of centrally administrated workstations at Iowa State University, patterned after Project Athena at MIT. With minor changes, the examples should carry over to settings outside of Project Vincent.

I have kept things simple rather than comprehensive. Some of my assertions may not even be quite correct, if I felt that a complete description would just confuse beginners. A lot of things are repeated in each relevant place, because I expect people to just read the parts that they are interested in.

The programming examples follow my particular tastes, which may or may not suit you. Feel free to adapt the examples to your style. For example, I strongly believe in declaring all variables in Fortran, and in prototyping all external subroutines in C. It is not strictly necessary, but it lets the compiler spot errors.

If you want to reproduce the case studies, you will need access to CMLIB and LAPACK. On Project Vincent, the following statements should be in your `.startup.tty` file:

```
add math
add lapack
```

If you are outside Project Vincent, see section *Finding a Routine* for more information on getting the source code for CMLIB or LAPACK.

The case studies for calling Fortran from C are

- DGAMMA (double precision gamma function) from FNLIB, which is part of CMLIB. This is the simplest case: a function with one input argument, one return value.
- DGESVD (double precision singular value decomposition) from LAPACK: Multiple arguments, some of them arrays.
- XERROR/XERBLA (error message routines from LINPACK/LAPACK): String arguments.
- DQAGS (double precision quadrature routine) from QUADPACK, which is part of CMLIB: Subroutine passed as argument.
- CGAMMA (complex gamma function). Same as DGAMMA, but for complex numbers.
- DGESVD Revisited. More details and suggestions for advanced programmers.

The sections on calling Fortran from Fortran and from C++ use a subset of the same examples.

Each section requires that you understand the material from previous sections, and that you are already familiar with Fortran, C, and C++. Some important concepts are repeated frequently, for the benefit of readers who only read part of this document.

2. Finding a Routine

Suppose you need a subroutine for a specific numerical task. Where do you start looking?
I would try the following things, in this order:

- Look on Project Vincent.

  "Wozu in die Ferne schweifen, wenn das Gute liegt so nah". If you find a suitable subroutine in some locker on Project Vincent, it will be already compiled and ready to go.

  The Math Department supports two main numerical packages: CMLIB and LAPACK. Documentation for both packages can be accessed through `http://www.math.iastate.edu/software.html`.

  CMLIB is a collection of many public domain sublibraries from Netlib, including LINPACK and EISPACK. Our version was compiled by Ronald F. Boisvert, Sally E. Howe and David K. Kahaner in April 1988. If you don't have CMLIB, you can get the necessary pieces from Netlib (see below).

  LAPACK is a linear algebra package intended to be efficient on many types of computer architecture. It can also be found on Netlib.

  CMLIB and LAPACK are the only two libraries illustrated here, but there are others in the `math` and `mathclasses` lockers.

  The Computation Center or other departments support other packages, for example the NAG library. Check `olc answers` for leads, try `which-locker` if you know the name of a package, look around the lockers of likely departments, or ask in a local newsgroup.
- Look in standard software repositories. The main ones I know of are
  - Netlib, accessible through `http://netlib.bell-labs.com/netlib/master/readme.html`
  - GAMS (Guide to Available Mathematica Software), accessible through `http://gams.nist.gov`.

  Whatever you find there needs to be compiled first.
- Ask for help.

  Ask a colleague, or post a question in a relevant newsgroup like `sci.math.num-analysis`. The turnaround time can be a little longer here, or you may not get a response, but you might get information you could not find on your own.

3. Calling Fortran from Fortran

If you want to reproduce the case studies in this section, you will need access to CMLIB and LAPACK. On Project Vincent, the following statements should be in your `.startup.tty` file:

```
add math
add lapack
```

Documentation for both packages can be found through `http://www.math.iastate.edu/software.html`. If you are outside Project Vincent, see section *Finding a Routine* for more information on getting the source code for CMLIB or LAPACK.

Linking Fortran to Fortran is very straightforward. You just have to tell the compiler where the libraries are located. Only one example for each library is provided.

Case Study 1: DGAMMA

This case study illustrates linking with CMLIB. CMLIB is located in the `math` locker.

We want to call the double precision gamma function DGAMMA from FNLIB, which is part of CMLIB, declared as

```
double precision function dgamma(x)
double precision x
...
```

This function calculates the gamma function $\Gamma(x)$.

The Fortran program `prog1.f` is

```
      program prog1
*
      double precision x, y, dgamma
*
      print*, 'enter x:'
      read*, x
      y = dgamma(x)
      print*, 'gamma(x) = ',y
      end
```

To compile and run the program:

```
% f77 -o prog1 prog1.f /home/math/lib/‘arch‘/libcm.a
% prog1
 enter x:
2.718
 gamma(x) =    1.56711274176688
```

An alternative way to specify linking with CMLIB is

```
% f77 -o prog1 prog1.f -L/home/math/lib/‘arch‘ -lcm
```

I usually use the first form for linking with a single library, the second form for linking with two or more libraries in the same directory.

If you type the command lines the way they are written, they will work on all three architectures currently supported on Project Vincent. You can also replace `‘arch‘` with `dec`, `axp` or `sgi`, depending on the machine you are using. These are backwards apostrophes around the `arch`; the font I am using does not show that very well.

Case Study 2: DGESVD

This case study illustrates linking with LAPACK. LAPACK is located in the `lapack` locker.

Routine DGESVD calculates the singular value decomposition (SVD) of a general double precision matrix. A matrix A of size $(m \times n)$, stored in array `A(lda,n)`, is decomposed into

$$A = U \Sigma V^T,$$

where U is an orthogonal matrix of size $(m \times m)$, stored in array `U(ldu,m)`, V^T is an orthogonal matrix of size $(n \times n)$, stored in array `VT(ldvt,n)`, and Σ is a diagonal matrix of size $(m \times n)$, with the diagonal stored in a one-dimensional array `S(min(m,n))`.

The declaration is

```
      subroutine dgesvd(jobu, jobvt, m, n, A, lda, S, U, ldu, VT, ldvt,
     +                  work, lwork, info )
      character         jobu, jobvt
      integer           info, lda, ldu, ldvt, lwork, m, n
      double precision  A(lda,*), S(*), U(ldu,*),
     +                  VT(ldvt,*), WORK(*)
```

For the exact meaning of all these parameters, consult the LAPACK documentation. All we need to know here is that `jobu = jobv = 'a'` will request that `U`, `VT` are actually calculated. (There are applications where `U`, `VT` are not needed, or only parts of them are needed; not calculating them saves time).

As an example, we will calculate the SVD of the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

The program is

```
      program prog2
*
      parameter (m=2, n=3, lwork=20)
      parameter (minmn=min(m,n))
      integer info
      double precision A(m,n), S(minmn), U(m,m), VT(n,n), work(lwork)
      data A /1,4,2,5,3,6/
*
      call dgesvd('a','a',m,n,A,m,S,U,m,VT,n,work,lwork,info)
      if (info .ne. 0) print*, 'DGESVD returned INFO =', info
      call print_matrix('U =',U,m,m)
      call print_matrix('Sigma =',S,1,minmn)
      call print_matrix('VT =',VT,n,n)
*
      end

      subroutine print_matrix(label,A,m,n)
      character*(*) label
      integer m, n, i, j
      double precision A(m,n)
*
      print*, label
      do 10 i = 1, m
         write (*,'(666f20.15)') (A(i,j),j=1,n)
 10   continue
*
```



```
      end
```

All of LAPACK is based on the BLAS (Basic Linear Algebra Subroutines). I have put the BLAS in a separate library so that you can use a different set of BLAS if you have one. You always need to link with both libraries.

We compile and run the program with

```
% f77 -o prog2 prog2.f -L/home/lapack/lib/`arch` -llapack -lblas
% prog2
 U =
  -0.386317703118612  -0.922365780077058
  -0.922365780077058   0.386317703118612
 Sigma =
   9.508032000695723   0.772869635673484
 VT =
  -0.428667133548626  -0.566306918848035  -0.703946704147444
   0.805963908589297   0.112382414096594  -0.581199080396110
   0.408248290463864  -0.816496580927726   0.408248290463863
```


4. Calling Fortran from C

This section contains some general guidelines on calling Fortran subroutines from a C main program. It is assumed that you already know C pretty well, and that you know how to do compiling and linking of Fortran or C programs.

If you want to reproduce the case studies in this section, you will need access to CMLIB and LAPACK. On Project Vincent, the following statements should be in your `.startup.tty` file:

```
add math
add lapack
```

Documentation for both packages can be found through `http://www.math.iastate.edu/software.html`. If you are outside Project Vincent, see section *Finding a Routine* for more information on getting the source code for CMLIB or LAPACK.

A Fortran routine is known to C by its name in lowercase, with underscore appended

Example: The Fortran subroutine `DGAMMA` is known to the C compiler as `dgamma_`.

All parameters need to be passed by reference

Fortran passes all arguments by *reference*. This means that the addresses of the actual arguments are passed to the subroutine. Anything the subroutine does to its arguments will affect the values of these variables in the main program.

C passes all arguments by *value*. This means that only copies of the values of the arguments are passed. Anything the subroutine does to its arguments will not affect the main program. The arguments are destroyed at the end of the subroutine call.

To imitate a call by reference from C, you need to pass *pointers* to the arguments, instead of the arguments itself. However, an array variable in C is already a pointer to the corresponding storage location, and you don't have to apply the referencing operator `&` a second time.

Note that this means that you can never pass a constant number directly to a Fortran subroutine. You have to assign the constant to a variable first, then pass the address of the variable.

Strings are arrays of characters, so you don't have to apply the referencing operator `&`, but you also have to pass the length of the string as a separate integer. See case study XERROR/XERBLA.

Example: A call by value in C:

```
float f(float x) {...}

main() {
   float x, y;
   y = f(x);
}
```

A call by reference in C:

```
float f(float *x) {...}

main() {
   float x, y;
   y = f(&x);
}
```

The Fortran subroutine

```
subroutine sub(x,y,z)
real x, y(10), z(10,10)
...
```

is called from C as


```
extern void sub_(float *x, float y[], float (*z)[]);

main() {
   float x, y[10], z[10][10];
   sub_(&x,y,z);
}
```

The declarations `float *x` and `float x[]` mean exactly the same: `x` is a pointer to a floating-point number. I use the first notation for `x` because it is a scalar, and the second for `y` because it is an array, but that is only for my personal benefit; the compiler doesn't care.

I use the declaration `float (*z)[]` because it works, while the more natural `float z[][]` does not. See the case study *DGESVD Revisited* for more information on this.

Arrays in Fortran are stored by column, arrays in C are stored by row. Subscript numbering in Fortran starts with 1, in C it starts with 0

A two-dimensional array

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

is strung out into a vector in memory. Fortran does it by columns, that is

$$a_{11}, a_{21}, \dots, a_{m1}, a_{12}, a_{22}, \dots$$

C does it by rows, that is

$$a_{11}, a_{12}, \dots, a_{1n}, a_{21}, a_{22}, \dots$$

If you pass a two-dimensional array between Fortran and C, one language will see the transpose of what the other language sees.

There are many ways to deal with this problem, depending on the individual circumstances:

- It may not matter at all.
- You may be able to rearrange calculations, for example by using $(AB)^T = B^T A^T$.
- Some LINPACK or LAPACK subroutines have a switch to make them use A^T instead of A.
- Maybe you really do need to transpose the array.

Do whatever works, just be aware of this problem.

There is no problem with one-dimensional arrays. Arrays of dimension higher than 2 are left as an exercise for the reader.

Also, keep in mind that the first entry in a Fortran array is `x(1)` or `A(1,1)`, in C it is `x[0]` or `A[0][0]`.

Do all your input/output in C

More specifically, in decreasing order of preference:

- The ideal case is when there are no read/write statements at all in the Fortran subroutine.
- The next best case is when there are some read/write statements in the subroutine, but they are not used (for example error messages that can be turned on or off).

  The program will run fine, but the presence of the Fortran I/O statements will cause the loader to link in 400K of Fortran I/O library code.

  If you have the Fortran source code, you may be able to comment out the read/write statements. Many subroutine libraries channel all I/O through a single routine; you may be able to rewrite that one routine in C. I have done this for XERROR from CMLIB and XERBLA from LAPACK; see case study *XERROR/XERBLA*.

- If you absolutely must have I/O in both main program and subroutine, use keyboard input/screen output, or use a different disk file in each language. That should still work ok.
- Do NOT read or write to the same disk file from both Fortran and C, unless it is closed and re-opened in between. You will get garbled input or output. You might even uncover some interesting flaws in Unix and get a garbled hard disk (unlikely, but conceivable).

Use the Fortran compiler for linking

There are two steps involved in producing a working program: compiling and linking. Unless you suppress linking with the `-c` compiler switch, the compiler will invoke the linker automatically and specify the location of the system libraries the linker needs.

For whatever internal reasons, the Fortran compiler knows where the C libraries are, but the C compiler does not know where the Fortran libraries are. You could figure out the exact location of all those libraries and specify them explicitly, but why bother? Just use the Fortran compiler for the linking step.

Corresponding Declarations

The following table gives a quick reference to corresponding data type declarations in Fortran and C. For explanations, read this overview section and/or the appropriate case study. Common blocks are mentioned in this table for completeness, but they are not mentioned anywhere else in the text; they don't usually come up in calling library subroutines.

Fortran	C
	`typedef struct { float re, im; } complex;` `typedef struct { double re, im; } double_complex;`

`integer x, y(10), z(10,20)`
`real`
`real*8`
`complex`
`complex*16`
	`int x, y[10], z[20][10];`
	`float`
	`double`
	`complex`
	`double_complex`

`character c` `character*10 s`	`char c, s[11];`

`real function f(x,y,z,c,s,f2)`
`real x,y(10),z(10,20), r`
`character c`
`character*20 s`
`real f2`
`external f2`
`r = f(x,y,z,c,s,f2)`
	`extern float f_(float *x, float y[], float (*z)[],`
	`char *c, char s[], float f2_(float *xx),`
	`int length_of_s);`
	`extern float f2_(float *x);`
	`main() {`
	`float x, y[10], z[20][10], r;`
	`char c, s[21];`
	`r = f_(&x,y,z,&c,s,f2_,strlen(s));`
	`}`

`common // a, b, c`
`common /cause/ a, b, c`
`real a, b(10)`
`integer c`
	`extern struct {float a; float b[10]; int c; } _BLNK__;`
	`extern struct {float a; float b[10]; int c; } cause_;`

Case Study 1: DGAMMA

This case study illustrates the simplest case: calling a simple function with one scalar argument and one return value.

We want to call the double precision gamma function DGAMMA from FNLIB, which is part of CMLIB, declared as

```
double precision function dgamma(x)
double precision x
...
```

This function calculates the gamma function $\Gamma(x)$.

The C program `dgamma.c` is

```
#include <stdio.h>
#include <math.h>

extern double dgamma_(double *x);

int main(int argc, char **argv) {
   double x, y;
   printf("enter x: ");
   scanf("%lf",&x);
   y = dgamma_(&x);
   printf("gamma(x) = %20.15lg\n",y);
   return 0;
}
```

To compile and run this program on a DecStation or SGI:

```
% f77 -o dgamma dgamma.c /home/math/lib/‘arch‘/libcm.a
% dgamma
enter x: 2.718
gamma(x) =     1.56711274176688
```

The Dec Alpha Fortran compiler is brain damaged and needs to be told explicitly that this program has NO FORtran MAIN program:

```
% f77 -nofor_main -o dgamma dgamma.c /home/math/lib/‘arch‘/libcm.a
% dgamma
enter x: 2.718
gamma(x) =     1.56711274176688
```

An alternative way to specify linking with CMLIB is

```
% f77 [-nofor_main] -o dgamma dgamma.c -L/home/math/lib/‘arch‘ -lcm
```

I usually use the first form for linking with a single library, the second form for linking with two or more libraries in the same directory. The brackets around `-nofor_main` mean: put this argument in if you are on an Alpha, otherwise leave it out.

If you type the command lines the way they are written, they will work on all three architectures currently supported on Project Vincent. You can also replace `‘arch‘` with `dec`, `axp` or `sgi`, depending on the machine you are using. These are backwards apostrophes around the `arch`; the font I am using does not show that very well.

You could compile `gamma.c` with the C compiler first, for esthetic reasons or because you want to use a different C compiler, but you should still use Fortran to do the linking:

```
% cc -c dgamma.c
% f77 [-nofor_main] -o dgamma dgamma.o /home/math/lib/‘arch‘/libcm.a
```


There is one more thing we can improve. The DGAMMA subroutine does not print anything, but it contains a call to a standard routine XERROR which produces error messages if necessary. That routine and thus the entire Fortran I/O library get linked in. We can reduce the size of the compiled program by about 400K by rewriting XERROR in C. See case study *XERROR/XERBLA*.

Case Study 2: DGESVD

This case study illustrates the use of one- and two-dimensional arrays as subroutine arguments.

Routine DGESVD calculates the singular value decomposition (SVD) of a general double precision matrix. A matrix A of size $(m \times n)$, stored in array `A(lda,n)`, is decomposed into

$$A = U \Sigma V^T,$$

where U is an orthogonal matrix of size $(m \times m)$, stored in array `U(ldu,m)`, V^T is an orthogonal matrix of size $(n \times n)$, stored in array `VT(ldvt,n)`, and Σ is a diagonal matrix of size $(m \times n)$, with the diagonal stored in a one-dimensional array `S(min(m,n))`.

The Fortran declaration is

```
      subroutine dgesvd(jobu, jobvt, m, n, A, lda, S, U, ldu, VT, ldvt,
     +                  work, lwork, info )
      character         jobu, jobvt
      integer           info, lda, ldu, ldvt, lwork, m, n
      double precision  A(lda,*), S(*), U(ldu,*), VT(ldvt,*), WORK(*)
```

For the exact meaning of all these parameters, consult the LAPACK documentation. All we need to know here is that `jobu = jobv = 'a'` will request that `U`, `VT` are actually calculated. (There are applications where `U`, `VT` are not needed, or only parts of them are needed; not calculating them saves time).

As an example, we will calculate the SVD of the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

The program is

```
#include <stdio.h>
#include <math.h>

extern void dgesvd_(char *jobu, char *jobvt, int *m, int *n,
                    double (*A)[], int *lda, double (*S)[], double (*U)[],
                    int *ldu, double (*VT)[], int *ldvt, double work[],
                    int *lwork, int *info);

void print_matrix(char *label, double (*A)[], int m, int n) {
  int i, j;
#define A(i,j) (*((double *)A + n*i + j))

  printf("%s\n",label);
  for (j=0; j<n; j++) {
    for (i=0; i<m; i++)
      printf("%20.15lf ",A(i,j));
    printf("\n");
  }
  printf("\n");
}

#define max(a,b) ((a) > (b) ? (a) : (b))
#define min(a,b) ((a) < (b) ? (a) : (b))

#define M 2
#define N 3
#define LWORK max(3*min(M,N)+max(M,N),5*min(M,N)-4)
```



```
int main(int argc, char **argv) {
  int    m = M, n = N, lwork = LWORK, info;
  double A[N][M] = {1,4,2,5,3,6},
         S[min(M,N)][1], U[M][M], VT[N][N], work[LWORK];
  char   job = 'a';

  dgesvd_(&job, &job, &m, &n, A, &m, S, U, &m, VT, &n, work, &lwork, &info);

  if (info != 0) fprintf(stderr,"DGESVD returned info = %d\n",info);

  print_matrix("U =",U,M,M);
  print_matrix("Sigma =",S,min(M,N),1);
  print_matrix("VT =",VT,N,N);

  return(0);
}
```

Observe that we are entering A^T as data, which is seen as A by DGESVD. Fortran returns U, V^T in Fortran ordering, which is seen as U^T, V by the C main program. Subroutine `print_matrix` actually prints the transpose of its argument, so the printout looks correct. `print_matrix` is explained in more detail in case study *DGESVD Revisited*.

I have declared `S` as a two-dimensional array so that I could call `print_matrix` on it. Normally, `S` should be one-dimensional.

I use the declaration `double (*A)[]` for a two-dimensional array because it works, while the more natural `double A[][]` does not. See case study *DGESVD Revisited* for more information on this.

Again, we include a rewritten error message routine to make the compiled program shorter (see case study *XERROR/XERBLA*).

We compile and run the program with

```
% f77 [-nofor_main] -o dgesvd dgesvd.c xerbla.c -L/home/lapack/lib/`arch` -llapack -lblas
% dgesvd
 U =
  -0.386317703118612  -0.922365780077058
  -0.922365780077058   0.386317703118612
 Sigma =
   9.508032000695723   0.772869635673484
 VT =
  -0.428667133548626  -0.566306918848035  -0.703946704147444
   0.805963908589297   0.112382414096594  -0.581199080396110
   0.408248290463864  -0.816496580927726   0.408248290463863
```


Case Study 3: XERROR/XERBLA

This case study illustrates the passing of strings as parameters. It is actually backwards compared to the other examples, because we are writing a C routine to be called from a Fortran library program. It still illustrates the point, and is more useful than a contrived example going the other way.

Subroutine packages typically channel all error messages through a single subroutine, so that messages can be turned off or diverted at a central place. If we rewrite this single subroutine in C, the entire Fortran I/O code does not need to be linked in, which saves about 400K of storage per compiled program.

I have done this here for routines XERROR from CMLIB, and XERBLA from LAPACK.

The definition of XERROR is

```
subroutine xerror(messg,nmessg,nerr,level)
character*72 messg
integer      nmessg, nerr, level
```

where `messg` is the error message itself, `nmessg` is the number of characters in the message, `nerr` the error number, and `level` the severity (2=fatal, 1=recoverable, 0=warning, -1=warning which is printed only once, even if XERROR gets called several times for this error).

Strings in both Fortran and C are arrays of characters. The difference is that Fortran keeps track of the actual length of a string in a separate (invisible, internal) integer variable, while C indicates the end of a string by a zero byte.

One consequence of this is that a C string must be one character longer than a corresponding Fortran string, to hold the zero byte. The other consequence is that Fortran must pass the length of the string as an invisible argument. This argument is passed by value, not by reference, and goes at the end of the argument list. If there are several string arguments, all length arguments go at the end, in the same order as the strings.

In this particular example the string length is passsed twice, in the hidden internal variable `messg_length` and also explicitly in `nmessg`. The subroutine could use either one. The unnecessary `nmessg` argument probably dates back to the days when it was a good policy not to assume too much about the intelligence of a Fortran compiler.

My XERROR routine does not do all the things the original does, and the formatting looks different, but it does get the error message printed. Implementing more features is left as an exercise for the reader.

```
#include <stdio.h>

void xerror_(char *messg, int *nmessg, int *nerr, int *level, int messg_length) {
  char message[73];
  int  i;

/* copy the message over into a C string */
  for (i = 0; i < *nmessg; i++) message[i] = messg[i];
  message[*nmessg] = '\0';

  switch (*level) {
  case 2:  fprintf(stderr,"Fatal Error ");
           break;
  case 1:  fprintf(stderr,"Recoverable Error ");
           break;
  default: fprintf(stderr,"Warning ");
  }

  fprintf(stderr,"%d\n%s\n",*nerr,message);
```



```
  exit(1);
}
```

Note how I am carefully copying over the message into a separate C string. The quick and dirty way is `messg[*nmessg] = '\0'`. However, if the error message is the full 72 characters long, this would write a zero byte over the following storage location, with unknown consequences.

It is unlikely that this will happen, and even if it does happen, it is unlikely that this would lead to any problems. Still, an error of this sort is virtually impossible to track down, and I do believe in defensive programming. Also, if you are paranoid: little errors like this have been exploited repeatedly by hackers.

Let's link DGAMMA first with the original XERROR routine from CMLIB, and test it on a DecStation:

```
% f77 -o dgamma dgamma.c /home/math/lib/‘arch‘/libcm.a
% ls -l dgamma
-rwxrwxr-x  1 keinert    478436 Nov  1 11:55 dgamma*
% dgamma
enter x: -1

 FATAL ERROR IN...
 DGAMMA  X IS A NEGATIVE INTEGER
 ERROR NUMBER =         4
 JOB ABORT DUE TO FATAL ERROR.
0          ERROR MESSAGE SUMMARY
 MESSAGE START             NERR     LEVEL     COUNT
 DGAMMA  X IS A NEGAT         4         2         1
```

With `xerror.c`

```
% f77 -o dgamma dgamma.c xerror.c /home/math/lib/‘arch‘/libcm.a
% ls -l dgamma
-rwxrwxr-x  1 keinert     81492 Nov  1 11:56 dgamma*
% dgamma
enter x: -1
Fatal Error 4
DGAMMA  X IS A NEGATIVE INTEGER
```

The program is now about 400K shorter, and still prints error messages.

To be honest, this trick is not really needed any longer on Alphas and SGIs. The newer machines all support dynamic linking. On a DEC Alpha, the file length can be shortened from 49152 to 32768, which isn't as dramatic a difference; on an SGI, the file size goes from 31196 to 24004. I assume there is still a difference of 400K or more of main memory during execution, when the dynamic libraries get loaded.

Subroutine XERBLA from LAPACK is much simpler, and reproduced in its entirety (minus comments):

```
      subroutine xerbla(srname,info)
      character*6        srname
      integer            info
*
      write(*,fmt =9999) srname, info
      stop
 9999 format(' ** On entry to ',a6,' parameter number ',i2,' had an illegal value')
      end
```

The C translation is

```
#include <stdio.h>

void xerbla_(char srname[6], int *info, int srname_length) {
  char name[7];
```



```
    int  i;

    for (i = 0; i < 6; i++) name[i] = srname[i];
    name[6] = '\0';
    fprintf(stderr," ** On entry to %6s parameter number %d had an illegal value\n",
                  name,*info);
    exit(1);
}
```

Normally, we would take the string length from `srname_length`, but in this particular case the LAPACK documentation states that all subroutine names have exactly 6 characters.

Case Study 4: DQAGS

This case study illustrates the passing of a subroutine name as an argument.

Routine DQAGS calculates an approximation to $\int_a^b f(x)\,dx$. The declaration is

```
      subroutine dqags(f,a,b,epsabs,epsrel,result,abserr,neval,ier,
     +                 limit,lenw,last,iwork,work)
      integer          neval, ier, limit, lenw, last, iwork(limit)
      double precision f, a, b, epsabs, epsrel, result, abserr, work(lenw)
      external         f
```

For the exact meaning of all these parameters, see the DQAGS documentation. DQAGS is part of QUADPACK, which is part of CMLIB. For our purposes it suffices to know that on input we have to specify the function `f`, integration limits `a`, `b` and the requested absolute and/or relative accuracy `epsabs`, `epsrel`. On return, we get the approximate integral `result`, the estimated actual error `abserr`, and the number of function evaluations done `neval`.

Whether the function to be integrated is written in Fortran or C makes no difference. You just have to make sure that a C function does argument passing by reference.

As an example, let's integrate e^x from 0 to 1 twice, once using a Fortran function and once using a C function.

```
#include <stdio.h>
#include <math.h>

extern void dqags_(double f(double *x), double *a, double *b, double *epsabs,
                   double *epsrel, double *result, double *abserr, int *neval,
                   int *ier, int *limit, int *lenw, int *last, int iwork[],
                   double work[]);

/* Fortran function */
extern double f1_(double *x);
/*    double precision function f1(x)
      double precision x
      f1 = exp(x)
      end
*/

/* C function */
double f2(double *x) {
  return exp(*x);
}

#define LIMIT 5
#define LENW  4*LIMIT

int main(int argc, char **argv) {
  int    neval, ier, limit=LIMIT, lenw=LENW, last, iwork[LIMIT];
  double a = 0., b = 1.e0, epsabs = 1.e-10, epsrel = 0., result, abserr, work[LENW];
  double exact = exp(1.e0) - 1;

  dqags_(f1_, &a, &b, &epsabs, &epsrel, &result, &abserr, &neval,
         &ier, &limit, &lenw, &last, iwork, work);
```



```
  if (ier > 0) {
    fprintf(stderr,"DQAGS returned IER = %d\n",ier);
  }
  printf("Fortran result  = %20.15lf\n",result);
  printf("estimated error = %20.15lf\n",abserr);
  printf("actual error    = %20.15lf\n",abs(exact - result));
  printf("computed using %d function evaluations\n\n",neval);

  dqags_(f2, &a, &b, &epsabs, &epsrel, &result, &abserr, &neval,
         &ier, &limit, &lenw, &last, iwork, work);
  if (ier > 0) {
    fprintf(stderr,"DQAGS returned IER = %d\n",ier);
  }
  printf("C result        = %20.15lf\n",result);
  printf("estimated error = %20.15lf\n",abserr);
  printf("actual error    = %20.15lf\n",abs(exact - result));
  printf("computed using %d function evaluations\n\n",neval);

  return(0);
}
```

We compile and run it with

```
% f77 [-nofor_main] -o dqags dqags.c f1.f xerror.c /home/math/lib/‘arch‘/libcm.a
% dqags
Fortran result  =    1.718281828459046
estimated error =    0.000000000000019
actual error    =    0.000000000000000
computed using 21 function evaluations

C result        =    1.718281828459046
estimated error =    0.000000000000019
actual error    =    0.000000000000000
computed using 21 function evaluations
```


Case Study 5: CGAMMA

This case study illustrates the use of complex numbers. This is not quite so simple. You should skip this example unless you really need to use complex numbers.

Routine CGAMMA is the complex equivalent of DGAMMA, that is, the gamma function $\Gamma(x)$ for complex arguments. It is declared as

```
complex function cgamma(x)
complex x
...
```

Note that CGAMMA is in single precision; I was not able to find a public domain double precision complex gamma function.

The main problem is that C does not have complex numbers. As far as storing them goes, that is easy to fix:

```
typedef struct { float  re, im; } complex;
typedef struct { double re, im; } double_complex;
```

Now you can have declarations like `complex x`, even arrays like `complex x[10]`, and they will be stored like Fortran complex numbers. You still don't get complex arithmetic. If you want to multiply complex numbers, you have to write your own multiplication routine, for example

```
complex cproduct(complex x, complex y) {
  complex z;
  z.re = x.re * y.re - x.im * y.im;
  z.im = x.re * y.im + x.im * y.re;
  return z;
}
```

Now you can program $z = x \cdot y$ as `z = cproduct(x,y)`, and similarly for other arithmetic operations. This notation gets tedious very fast.

If you really want to work with complex numbers in C, it may be worthwhile learning C++. C++ lets you define arithmetic operations on new data types using standard notation, and I am sure this has been done already for complex numbers.

An even bigger problem occurs with functions that return a complex number, like CGAMMA. At first, the interface appears straightforward, just a minor modification of the DGAMMA example:

```
#include <stdio.h>
#include <math.h>

typedef struct { float  re, im; } complex;
extern complex cgamma_(complex *x);

int main(int argc, char **argv) {
  complex x, y;
  printf("enter x: ");
  scanf("%f%f",&(x.re),&(x.im));
  y = cgamma_(&x);
  printf("gamma(x) = %10.7g + i %10.7g\n",y.re,y.im);
  return 0;
}
```

This program compiles and runs, but produces garbage.

In Fortran, functions can return only scalars. Real scalars are returned in one particular CPU register, complex scalars use two registers.

In C, functions can return scalars, pointers, structures, maybe even arrays. Anything that fits into one CPU register (a real number, integer, character, or pointer) is returned in the same CPU register that Fortran uses. Anything longer is returned on the stack.

The one place where these conventions clash is complex numbers: Fortran returns a complex number in two registers, C uses the stack.

I don't see how you could access a Fortran complex return value from C without using assembly language. The best easy solution I came up with is to create a Fortran interface routine CGAMMA2 which converts a function return value into a subroutine argument:

```
subroutine cgamma2(x,y)
complex x, y, cgamma
y = cgamma(x)
end
```

Now we can get to the return value `y` from C:

```
#include <stdio.h>
#include <math.h>

typedef struct { float  re, im; } complex;
extern void cgamma2_(complex *x, complex *y);

int main(int argc, char **argv) {
  complex x, y;
  printf("enter x: ");
  scanf("%f%f",&(x.re),&(x.im));
  cgamma2_(&x,&y);
  printf("gamma(x) = %10.7g + i %10.7g\n",y.re,y.im);
  return 0;
}
```

To compile and run

```
% f77 [-nofor_main] -o cgamma cgamma.c cgamma2.f xerror.c /home/math/lib/‘arch‘/libcm.a
% cgamma
enter x: 2 1
gamma(x) =  0.6529655 + i   0.343066
```


Case Study 6: DGESVD Revisited

This case study deals with advanced topics. Most users should skip this section.

Two-Dimensional Arrays

Let's look at the reason why a two-dimensional array argument is declared as `double (*A)[]` instead of the more natural `double A[][]`, and why my `print_matrix` subroutine in case study *DGESVD* is so messy. We need to consider how array subscripting is implemented.

Two-dimensional arrays in C are stored by rows. That means that if `A` is declared as

```
double A[5][10];
```

then `A[i][j]` refers to the location `(10*i+j)` from the start. In order to do address calculations correctly, the compiler needs to know the second dimension of `A`. Similarly, the Fortran compiler needs to know the first dimension.

When a two-dimensional array is passed to a subroutine, the dimension information must be made available to the subroutine. Fortran allows dimensions to be passed as a parameters, as in

```
subroutine sub(A,m,n)
integer m, n
real    A(m,n)
```

As far as I know, neither C nor C++ let you do this. The only solution is to do your own address arithmetic. In `print_matrix`, I am doing that in a macro, to keep the code halfway readable.

As far as declaring the argument type goes, C rejects `double A[][]` since it contains no dimension information. It would accept `double A[][10]` (only the second dimension is essential), but then you could only pass arrays with a second dimension of 10. Some experimentation determined that the compiler will accept the notation `double (*A)[]`, which means that `A` is a pointer to an array of doubles.

A Fancier Interface

Let's assume you want to do a whole bunch of work with routines from some Fortran library. It would be nice to hide all the messy details in some interface routines once and for all. Here is how I would go about it. Note the similarity with the Matlab interface in my other document *Calling Fortran from Matlab*.

After looking at a couple of matrix packages for C and C++, I decided that I like Matlab's approach best. My `matrix` data structure is a stripped-down version of what Matlab uses internally. All sizing information is contained in the matrix itself, and matrices (in particular working storage) can be created and destroyed as needed. These matrices are stored by columns, so we can interface them more easily to Fortran.

My interface routine `dgesvd` takes 4 parameters: input matrix `A`, output matrices `U`, `S`, `VT`. It takes care of setting up the other 10 parameters `dgesvd_` requires. In detail, it does the following:

- extract dimension information from the arguments
- create working storage
- call `dgesvd_`
- destroy working storage

I will let the code speak for itself:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define max(a,b) ((a) > (b) ? (a) : (b))
#define min(a,b) ((a) < (b) ? (a) : (b))

typedef struct { int m, n; double *data; } matrix;

matrix *create_matrix(int m, int n) {
  matrix *M;
```



```
  M = (matrix *) malloc(sizeof(matrix));
  if (M == NULL) {
    fprintf(stderr,"Memory allocation error\n");
    exit(1);
  }

  M->m = m;
  M->n = n;

  M->data = (double *) malloc(m * n * sizeof(double));
  if (M->data == NULL) {
    fprintf(stderr,"Memory allocation error\n");
    exit(1);
  }

  return M;
}

void destroy_matrix(matrix *M) {
  free(M->data);
  free(M);
}

void print_matrix(char *label, matrix *A) {
  int i, j;
#define A(i,j) (*(A->data + (A->m)*j + i))

  printf("%s\n",label);
  for (i = 0; i < A->m; i++) {
    for (j = 0; j < A->n; j++)
      printf("%20.15lf ",A(i,j));
    printf("\n");
  }
  printf("\n");
}

extern void dgesvd_(char *jobu, char *jobvt, int *m, int *n,
                    double A[], int *lda, double S[], double U[],
                    int *ldu, double VT[], int *ldvt, double work[],
                    int *lwork, int *info);

int dgesvd(matrix *A, matrix *U, matrix *S, matrix *VT) {
  char   job = 'a';
  int    m = A->m, n = A->n, lwork = 5*max(m,n), info;
  matrix *work;

  work = create_matrix(lwork,1);

  dgesvd_(&job,&job,&m,&n,A->data,&(A->m),S->data,U->data,&(U->n),
          VT->data,&(VT->n),work->data,&lwork,&info);
```



```
  destroy_matrix(work);

  return(info);
}

#define M 2
#define N 3

int main(int argc, char **argv) {
  double data[] = {1,4,2,5,3,6};
  matrix *A  = create_matrix(M,N),
         *U  = create_matrix(M,M),
         *VT = create_matrix(N,N),
         *S  = create_matrix(min(M,N),1);
  int    info;

  A->data = data;

  print_matrix("A =",A);

  info = dgesvd(A, U,S,VT);

  if (info != 0) fprintf(stderr,"DGESVD returned info = %d\n",info);

  print_matrix("A =",A);
  print_matrix("U =",U);
  print_matrix("Sigma =",S);
  print_matrix("VT =",VT);

  return(0);
}
```

The program compiles and runs like before.

5. Calling Fortran from C++

This section describes how to call a Fortran subroutine from C++. It is very rudimentary because I know very little about C++. One of these days I hope to expand the section and/or integrate it better with the section on C, but don't hold your breath. It is assumed that you fully understand the section *Calling Fortran from C*.

If you want to reproduce the case study in this section, you will need access to CMLIB and the Gnu C++ compiler. On Project Vincent, the following statements should be in your `.startup.tty` file:

```
add math
add gcc
```

Documentation for CMLIB can be found through `http://www.math.iastate.edu/software.html`. The official documentation for the Gnu C++ compiler is a hypertext document available from inside the Emacs editor. You can access it by pressing `Ctrl-U Ctrl-H i` in Emacs, and giving `/home/gcc2/info/gcc.info` as the document name. There is a `man` page for C++, but it states that it may be out of date and will not be updated any more. Take your chances if you want.

All the guidelines from the section *Calling Fortran from C* apply also to C++, and there is one new one:

Wrap `extern "C"` {...} around the `extern` statements referring to Fortran or plain C

One of the main features of C++ is *overloading*. This means that several subroutines can share the same name, as long as they have different types of arguments. The compiler will figure out from the arguments which of the routines you mean. This is not unlike the situation in Fortran, where `y = exp(x)` invokes the single precision, double precision, or complex exponential function, depending on the type of `x`.

In order to maintain its sanity, the C++ compiler internally appends a string indicating the argument types to each subroutine name. For the Fortran interface, you need to turn this feature off. You do this by declaring your Fortran or plain C subroutines inside an `extern "C"` {...} environment. See case study *DGAMMA* for an example.

Case Study 1: DGAMMA

This case study illustrates mainly the linking step. For details on the interface, see section *Calling Fortran from C*. As an example, we use the simplest case: calling a simple function with one scalar argument and one return value.

We want to interface the double precision gamma function DGAMMA from FNLIB, which is part of CMLIB, declared as

```
double precision function dgamma(x)
double precision x
...
```

This function calculates the gamma function $\Gamma(x)$.

The C++ program `dgamma.C` is

```
#include <stream.h>
#include <math.h>

extern "C" {
  extern double dgamma_(double *x);
}

int main(int argc, char **argv) {
  double x, y;
  cout << "enter x: ";
  cin >> x;
  y = dgamma_(&x);
  cout << "gamma(x) = " << y << "\n";
  return 0;
}
```

This is basically the same as `dgamma.c` in section *Calling Fortran from C*, except for the wrapper around the `extern` declaration and the different I/O statements.

Linking is a real headache. As mentioned before, the linking must be done by the Fortran compiler; unfortunately, the Fortran compiler knows where the system C libraries are, but not where the C++ libraries are. That is because the system C compiler and the system Fortran compiler come from the same company, the C++ compiler does not. The situation is complicated by the fact that the C++ libraries are located in directories with very long, complicated names.

The following instructions are accuracte as of October 22, 1996. As soon as a new version of the Gnu C++ compiler gets installed, or the libraries get moved for some other reason, this part will break.

If you need to do this from scratch, you can figure out the system libraries as follows: write a trivial C++ program `hello.C`

```
#include <stream.h>

int main(int argc, char **argv) {
  cout << "hello, world\n";
  return 0;
}
```

and compile and link it verbosely. Only the `-L` and `-l` arguments in the `ld` statement at the end are important. On my DecStation at the time of writing, I got the following (after splitting the line across several lines, for readability):

```
% g++ -v hello.C
...
/afs/iastate.edu/project/gcc2/ultrix43a/lib/gcc-lib/mips-dec-ultrix4.3/2.7.2/ld
```



```
  /usr/lib/cmplrs/cc/crt0.o
  -L/afs/iastate.edu/project/gcc2/ultrix43a/lib/gcc-lib/mips-dec-ultrix4.3/2.7.2
  -L/usr/lib/cmplrs/cc
  -L/usr/lib/cmplrs/cc
  -L/afs/iastate.edu/project/gcc2/ultrix43a/lib
  /usr/tmp/cca240561.o -lg++ -lstdc++ -lm -lgcc -lc -lgcc
%
```

Anything in `/usr/lib` would be known to the Fortran compiler, and `/afs/iastate.edu/project/gcc2` is the same as `/home/gcc2`, so we need to link with

```
-L/home/gcc2/ultrix43a/lib/gcc-lib/mips-dec-ultrix4.3/2.7.2
-L/home/gcc2/ultrix43a/lib -lg++ -lstdc++ -lm -lgcc -lc -lgcc
```

Now we are ready:

```
% g++ -c dgamma.C
% f77 -o dgamma dgamma.o /home/math/lib/dec/libcm.a
      -L/home/gcc2/ultrix43a/lib/gcc-lib/mips-dec-ultrix4.3/2.7.2
      -L/home/gcc2/ultrix43a/lib -lg++ -lstdc++ -lm -lgcc -lc -lgcc
% dgamma
enter x: 2.718
gamma(x) = 1.56711
%
```

Everything after the `f77` is on one huge line; I have split it here only for readability.

On an Alpha, the linking command is

```
% f77 -nofor_main -o dgamma dgamma.o /home/math/lib/axp/libcm.a
      -L/home/gcc2/du3.2/lib/gcc-lib/alpha-dec-osf3.2/2.7.2
      -L/home/gcc2/du3.2/lib -lg++ -lstdc++ -lm -lgcc -lc -lgcc
```

On an SGI, the command is

```
% f77 -o dgamma dgamma.o /home/math/lib/sgi/libcm.a
      -L/home/gcc2/sgi53/lib/gcc-lib/mips-sgi-irix5.3/2.7.2
      -L/home/gcc2/sgi53/lib -lg++ -lstdc++ -lm -lgcc -lc -lgcc
```

To keep from going insane, you probably want to put those library names in some environment variable, alias, or shell script. For example, you could put the following in your `.cshrc.mine` file:

```
if (`arch` == "dec") then
    setenv GCCLIBS "-L/home/gcc2/ultrix43a/lib/gcc-lib/mips-dec-ultrix4.3/2.7.2
                    -L/home/gcc2/ultrix43a/lib -lg++ -lstdc++ -lm -lgcc -lc -lgcc"
else if (`arch` == "axp") then
    setenv GCCLIBS "-L/home/gcc2/du3.2/lib/gcc-lib/alpha-dec-osf3.2/2.7.2
                    -L/home/gcc2/du3.2/lib -lg++ -lstdc++ -lm -lgcc -lc -lgcc"
else
    setenv GCCLIBS "-L/home/gcc2/sgi53/lib/gcc-lib/mips-sgi-irix5.3/2.7.2
                    -L/home/gcc2/sgi53/lib -lg++ -lstdc++ -lm -lgcc -lc -lgcc"
endif
```

(again, all the `setenv` statements should be one line, not two). Then you can compile with

```
% g++ -c dgamma.C
% f77 [-nofor_main] -o dgamma dgamma.o /home/math/lib/dec/libcm.a $GCCLIBS
```

On Alphas running Digital UNIX 3.2D, a C++ compiler from DEC is now available. Unfortunately, it is not as nicely integrated with Fortran as the standard C compiler. You still have to specify a bunch of libraries to link with. The linking sequence for our example is

```
% cxx -c dgamma.C
```



```
% f77 -nofor_main -o dgamma dgamma.o /home/math/lib/dec/libcm.a
      /usr/lib/cmplrs/cxx/_main.o  -L/usr/lib/cmplrs/cxx -lcxxstd -lcxx -lexc
% dgamma
enter x: 2.718
gamma(x) = 1.56711
```


E-mail address:

